Waldemar Celes
Renato Cerqueira
José Lucas Rangel

Introdução a Estruturas de Dados

Com técnicas de programação em C

Waldemar Celes
Renato Cerqueira
José Lucas Rangel

Introdução a Estruturas de Dados

Com técnicas de programação em C

4ª Tiragem

Sumário

Prefácio

O conhecimento de técnicas de programação adequadas para a elaboração de programas de computador tornou-se indispensável para profissionais que atuam nas áreas técnico-científicas. Por essa razão, o ensino de programação tornou-se um requisito básico para a formação desses profissionais. Muitos deles estarão diretamente envolvidos com o desenvolvimento de software e, portanto, precisam de um profundo conhecimento de técnicas de programação; outros serão usuários de software desenvolvidos para atender a requisitos específicos das áreas em que atuam. No entanto, mesmo para esses profissionais, um conhecimento adequado de programação se faz necessário, seja para tirar proveito do acesso programável que as atuais aplicações oferecem, seja para ajudar na avaliação da qualidade dos programas apresentados.

O conhecimento de uma linguagem de programação por si só não capacita programadores, pois é necessário saber usar os recursos de programação de maneira adequada. A elaboração de um programa envolve diversas etapas, incluindo a identificação das propriedades dos dados e suas características funcionais. Para que possamos fazer um programa atender de maneira eficiente às funcionalidades para as quais ele foi projetado, precisamos conhecer técnicas para organizar de maneira estruturada os dados a serem manipulados. Assim, além de uma linguagem de programação, precisamos conhecer as principais técnicas de estruturação de dados.

O objetivo deste livro é apresentar aos leitores os conceitos básicos de estruturas de dados. Para isso, optamos por uma abordagem bastante prática, discutindo as funcionalidades das estruturas de dados com base na sua implementação em programas de exemplo. Dessa forma, esperamos que os leitores tenham uma visão prática das estruturas e consigam facilmente adaptá-las a aplicações específicas de seu interesse.

Para a apresentação das estruturas de dados, optamos por usar a linguagem de programação C. Apesar de reconhecer as dificuldades em sua aprendizagem, optamos por sua utilização simplesmente porque C é a linguagem básica de programação de maior uso atualmente. Um ponto adicional a favor da escolha de C é a facilidade na aprendizagem de qualquer outra linguagem de programação, incluindo as linguagens orientadas a objetos, como C++ e Java, se programamos em C com desenvoltura.

Este livro visa a atender às demandas de cursos introdutórios de programação, seja para alunos de cursos na área de Informática, seja para alunos nas mais diversas áreas técnico-científicas, tais quais Engenharia, Matemática e Física. O livro abrange um conteúdo que em geral é apresentado numa segunda ou terceira disciplina de Informática. A primeira parte do livro também pode servir de referência para um curso introdutório da linguagem de programação C.

Também esperamos que o livro cumpra seu objetivo de servir como referência para profissionais já formados que necessitem aprender ou recapitular os conceitos de programação em C e o uso das estruturas de dados básicas.

O conteúdo do livro está dividido em três partes. A Parte I exibe as estruturas de dados que convencionamos chamar de estáticas, construídas sobre as formas simples de estruturação de dados oferecidas pelas linguagens de programação, como vetores e tipos estruturados. Nos primeiros capítulos da Parte I, optamos por mostrar os conceitos fundamentais da linguagem de programação em C, facilitando o acesso à discussão sobre as estruturas de dados para os leitores que ainda não conhecem ou que têm pouco conhecimento de C. A Parte II apresenta as estruturas de dados dinâmicas, tais como listas encadeadas e árvores, que oferecem um suporte mais adequado para a inserção e remoção de elementos dinamicamente. Ao final dessa segunda parte, discutimos a elaboração de estruturas de dados genéricas, as quais podem ser utilizadas para armazenar qualquer tipo de dado. Finalmente, a Parte III do livro discute os algoritmos de ordenação e busca, e expõe estruturas de dados projetadas especificamente para realizar de forma mais eficiente essas operações, as quais são comumente necessárias para o desenvolvimento de diversas aplicações computacionais. Essa terceira parte também discute o desenvolvimento e a utilização de algoritmos genéricos que podem operar sobre um conjunto de dados de qualquer tipo.

PARTE I

Estruturas estáticas

Este livro discute as estruturas de dados básicas, e apresenta diversas técnicas de programação que podem ser usadas no desenvolvimento de programas de computador. Para a implementação dessas estruturas de dados, optamos por trabalhar com a linguagem de programação C. Ela tem sido amplamente utilizada na elaboração de programas e sistemas nas diversas áreas em que a informática atua, e seu aprendizado tornou-se indispensável para quem trabalha com programação de computadores. Por isso, a linguagem C vem sendo muito utilizada para o ensino de programação.

Esta primeira parte do livro revela as estruturas de dados que convencionamos chamar de estáticas, pois não oferecem suporte adequado para a inserção e remoção de elementos dinamicamente. Essas estruturas são baseadas na utilização de formas primitivas de estruturação de dados disponibilizadas pela linguagem de programação, como vetores e tipos estruturados.

Nos primeiros três capítulos, abordamos os conceitos básicos da linguagem de programação C. No quarto capítulo, discutimos em detalhes a construção de funções e a forma de comunicação entre elas, estudando conceitos importantes para a implementação de estruturas de dados, como o tempo de vida e o escopo de variáveis locais. O leitor já familiarizado com C pode omitir a leitura desses capítulos iniciais.

A partir do Capítulo 5, são apresentadas as estruturas de dados estáticas. Inicialmente, discutimos a utilização de vetores e introduzimos o conceito de alocação dinâmica de memória. A seguir, no Capítulo 6, vemos a representação de conjuntos bidimensionais (matrizes) e apontamos diferentes estratégias para tratar matrizes alocadas dinamicamente. O Capítulo 7 discute a representação de cadeias de caracteres em C e, por fim, o Capítulo 8 demonstra formas estruturadas para representarmos dados complexos.

1

Conceitos fundamentais

A linguagem C, assim como Pascal e Fortran, é considerada uma linguagem de programação "convencional". Para programar em uma linguagem convencional, precisamos, de alguma maneira, especificar as áreas de memória em que os dados com os quais queremos trabalhar estão armazenados e, freqüentemente, considerar os endereços de memória em que estão os dados. Isso faz o processo de programação envolver detalhes adicionais, possíveis de serem ignorados em uma linguagem de nível mais alto, como as linguagens funcionais e as linguagens de script. Em compensação, temos um maior controle da máquina quando utilizamos uma linguagem convencional e podemos fazer programas melhores do ponto de vista do uso dos recursos computacionais, ou seja, menores e mais rápidos.

A linguagem C provê as construções de controle de fluxo fundamentais para programas bem estruturados: agrupamentos de comandos, tomadas de decisão (*if-else*), laços com testes de encerramento no início (*while*, *for*) ou no fim (*do-while*) e seleção de um caso entre um conjunto de casos possíveis (*switch*). Ela oferece ainda o acesso a endereços de variáveis e a capacidade de fazer aritmética com esses endereços. Por outro lado, não provê operações para manipular diretamente objetos compostos, como cadeias de caracteres, nem facilidades de entrada e saída: não há comandos específicos para a entrada ou a saída de dados, por exemplo. Todos esses mecanismos devem ser fornecidos por funções explicitamente chamadas. Embora a falta de algumas dessas facilidades possa parecer uma deficiência grave (deve-se, por exemplo, chamar uma função para comparar duas cadeias de caracteres), a manutenção da linguagem em termos modestos tem trazido benefícios reais. A linguagem C é relativamente pequena e, no entanto, tornou-se muito poderosa e eficiente.

Modelo de um computador

Existem diversos tipos de computadores. Embora não seja nosso objetivo estudar hardware, nesta seção identificaremos os elementos essenciais de um computador. O conhecimento desses elementos nos ajudará a compreender como um programa de computador funciona.

A Figura 1.1 identifica os elementos básicos de um computador típico. O canal de comunicação (conhecido como BUS) representa o meio para a transferência de dados entre os diversos componentes. A unidade central de processamento (CPU) representa o "cérebro" do computador, e é responsável pelo controle de todas as operações realizadas. Para que a CPU possa executar uma seqüência de comandos ou acessar uma determinada informação, é necessário armazenar os comandos e os dados correspondentes na memória principal. Nela são armazenados, portanto, os programas e os dados manipulados pela CPU. A memória principal tem acesso randômico, o que significa que a CPU pode endereçar (isto é, acessar) diretamente qualquer posição da memória. Essa memória não é permanente e, para um programa, os dados são armazenados enquanto ele está sendo executado. Normalmente, após o término do programa, a área correspondente ocupada na memória fica disponível para ser usada por outros programas.

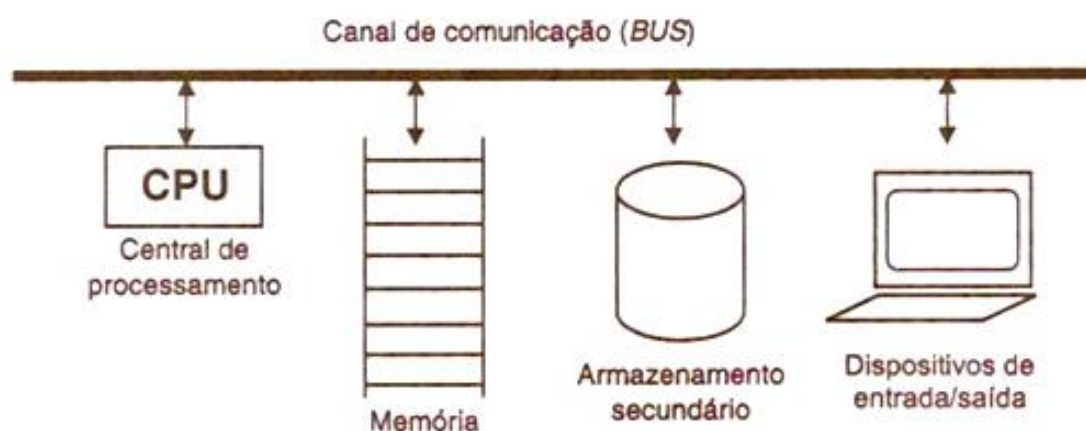



Figura 1.1 *Elementos básicos de um computador típico.*

A área de armazenamento secundário é, em geral, representada por meios magnéticos (disco rígido, disquete etc.). Essa memória secundária tem a vantagem de ser permanente. Os dados armazenados em disco permanecem válidos mesmo depois do encerramento dos programas. Ela tem um custo mais baixo do que a memória principal, porém o acesso aos dados é bem mais lento. Para que a CPU processe um dado armazenado na memória secundária, é necessário que antes ele seja transferido para a memória principal.

Por fim, encontram-se os dispositivos de entrada e saída. Os dispositivos de entrada (por exemplo, teclado, mouse) permitem passar dados para um programa, enquanto os dispositivos de saída permitem que um programa exporte seus resultados, por exemplo em forma textual ou gráfica, usando monitores ou impressoras.

Armazenamento de dados e programas na memória

A memória do computador é dividida em unidades de armazenamento chamadas bytes. Cada byte é composto por 8 bits. Cada posição da memória (byte) tem um endereço único. Não é possível endereçar diretamente um bit. Cada bit pode armazenar o valor zero (desligado ou desativado) ou um (ligado ou ativado). Nada além de zeros e uns pode ser armazenado na memória do computador. Por essa razão, todas as informações (programas, textos, imagens etc.) são armazenadas com o uso de uma codificação numérica na forma binária. Na representação binária, os números são representados por uma seqüência de zeros e uns. No nosso dia-a-dia, usamos a representação decimal, ou seja, representamos os números com 10 algarismos, de 0 a 9.

Da mesma maneira que podemos representar os números no nosso sistema com dez algarismos, podemos também representar números na base binária, isto é, com apenas dois algarismos. Por exemplo, na base decimal, o número 456 representa o valor $4*10^2 + 5*10^1 + 6*10^0$. O algarismo da centena (4) é multiplicado pela base (10) elevada ao expoente da centena (2); o algarismo da dezena (5) é multiplicado pela base (10) elevada ao expoente da dezena (1), e assim por diante. De forma análoga, podemos representar um número na base binária. Por exemplo, o número 101 na base binária representa o número decimal 5, pois $1*2^2 + 0*2^1 + 1*2^0$ é igual a 5.

Como veremos no próximo capítulo, quando reservamos um espaço de memória para armazenar um determinado valor, esse espaço é finito, composto de 1 ou mais bytes. Portanto, a faixa de valores e a precisão com que representamos um valor no computador são finitas, pois temos um número finito de bits para essa representação. Assim, em um espaço de 1 byte (8 bits), só podemos representar 2^8 (= 256) valores distintos.

Se só podemos armazenar números na memória do computador, como fazemos para armazenar um texto (um documento ou uma mensagem)? Para armazenar uma seqüência de caracteres, que representa o texto, atribui-se a cada caractere um código numérico (por exemplo, pode-se associar ao caractere A o código 65, ao caractere B o código 66, e daí por diante). Se todos os caracteres tiverem códigos associados (inclusive os caracteres de pontuação e de formatação), podemos armazenar um texto na memória do computador como uma seqüência de códigos numéricos.

A mesma estratégia é usada para representar um programa na memória do computador. Um computador só pode executar programas em linguagens de máquina. Cada programa executável é uma seqüência de instruções que o processador central interpreta, executando as operações correspondentes. Essa seqüência de instruções também é representada como uma seqüência de códigos numéricos. Os programas ficam armazenados em disco e, para serem executados pelo computador, devem ser carregados (transferidos) para a memória principal. Uma vez na memória, o computador executa a seqüência de operações correspondente.

Compilação de programas

Ao escrever um programa, estamos codificando uma seqüência de operações para serem executadas pelo computador. No entanto, um programa escrito em C não pode ser diretamente executado, pois os computadores só executam programas em sua linguagem de máquina (à qual vamos nos referir como M), específica a cada modelo (ou família de modelos) de computador.

C é uma linguagem compilada, o que significa que um programa escrito em C (P_C) só pode ser executado se antes for "traduzido" para a linguagem de máquina correspondente ao modelo do computador usado. A esse processo damos o nome de *compilação*. Um programa compilador (C_M), escrito em M, lê o programa P_C, escrito em C, e traduz cada uma de suas instruções para M, escrevendo um programa objeto P_M cujo efeito é o desejado. Como conseqüência desse processo, P_M, por ser um programa escrito em M, pode ser executado em qualquer máquina com a mesma linguagem de máquina M. A máquina em que o programa é executado não precisa ter um compilador instalado nem precisa ter acesso ao código C do programa.

Dessa forma, a construção de um programa que usa a linguagem C envolve duas fases independentes: compilação e execução, conforme ilustra a Figura 1.2. Na primeira fase, o programa objeto é a saída do programa compilador; na segunda, o programa objeto é executado, recebendo os dados de entrada e gerando a saída correspondente.

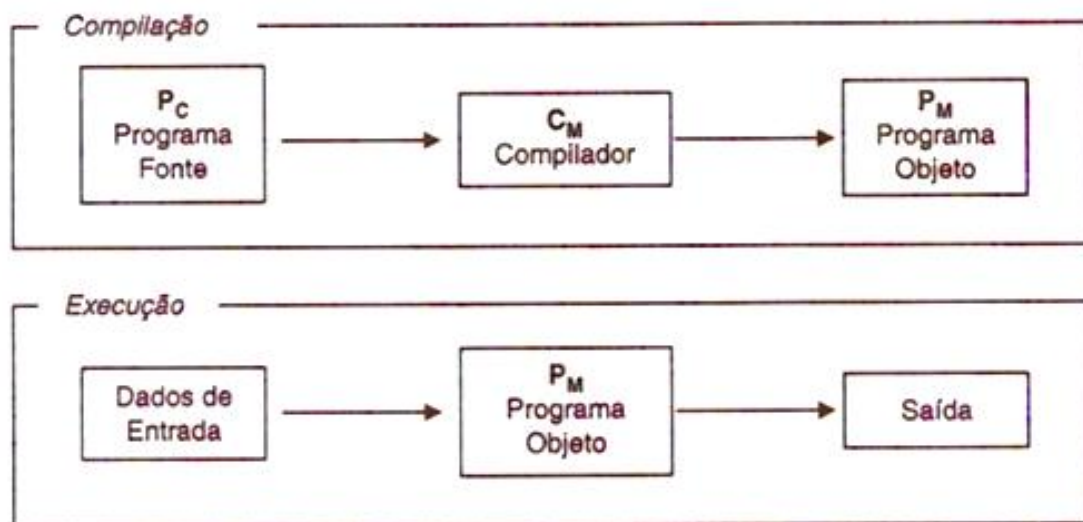



Figura 1.2 *Execução de programas com linguagem compilada.*

Na prática, o programa-fonte e o programa objeto são armazenados em arquivos em disco, aos quais nos referimos como arquivo fonte e arquivo objeto.

O termo "máquina" usado anteriormente é intencionalmente vago. Por exemplo, computadores idênticos com sistemas operacionais diferentes devem ser considerados "máquinas", ou "plataformas", diferentes. Assim, um progra-

ma em C compilado em um PC com Windows não pode ser executado em um PC com Linux e vice-versa. Para cada "máquina", devemos repetir o processo de compilação.

Exemplo de código em C

Para exemplificar códigos escritos em C, apresentaremos o código de um programa simples que converte temperaturas fornecidas em graus Celsius para Fahrenheit. O leitor não familiarizado com a linguagem C não deve se preocupar com a compreensão do programa mostrado. Todas as características essenciais da linguagem C serão apresentadas e discutidas em detalhes nos capítulos subseqüentes. O objetivo de mostrar o programa agora é simplesmente apresentar a forma dos programas escritos em C e discutir alguns aspectos gerais da organização do código.

Esse programa para a conversão de temperatura define uma função principal que captura um valor de temperatura em Celsius, fornecido via teclado pelo usuário, e exibe como saída a temperatura correspondente em Fahrenheit. Para fazer essa conversão, é utilizada uma função auxiliar. O código C desse programa exemplo é mostrado a seguir.

```
/* Programa para conversão de temperatura */

#include <stdio.h>

/* Função auxiliar */
float converte (float c)
{
  float f;
  f = 1.8*c + 32;
  return f;
}

/* Função principal */
int main (void)
{
  float t1;
  float t2;
  /* mostra mensagem para usuário */
  printf("Digite a temperatura em Celsius: ");
  /* captura valor entrado via teclado */
  scanf("%f",&t1);
  /* faz a conversão, chamando função auxiliar */
  t2 = converte(t1);
  /* exibe resultado */
  printf("Temperatura em Fahrenheit: %f\n", t2);
  return 0;
}
```

A linguagem C não impõe o uso de uma formatação rígida. Nela, o programador escolhe a forma mais apropriada para escrever seu código. Podemos, por exemplo, escrever vários comandos em uma única linha ou dividir um mesmo comando em diversas linhas. No entanto, para que nossos códigos tenham clareza, na maioria das vezes optamos por escrever cada comando em uma linha, sempre que possível.

Um programa em C, em geral, é constituído de diversas funções pequenas, independentes entre si. Não podemos, por exemplo, definir uma função dentro de outra. Dois tipos de ambientes são caracterizados em um código C: o ambiente global, externo às funções, e os ambientes locais, definidos pelas diversas funções (lembrando que os ambientes locais são independentes entre si). Pode-se inserir comentários no código-fonte, iniciados com /* e finalizados com */, conforme ilustrado anteriormente. Devemos notar também que comandos e declarações em C são terminados pelo caractere ponto-e-vírgula (`;`).

Um programa em C tem de, obrigatoriamente, conter a função principal (`main`), uma vez que a execução de um programa começa sempre por ela. A função `main` é automaticamente chamada quando o programa é carregado para a memória. As funções auxiliares são chamadas, direta ou indiretamente, a partir da função principal.

Em C, como nas demais linguagens "convencionais", devemos reservar uma área na memória para armazenar cada dado. Isso é feito usando a declaração de variáveis, na qual informamos o tipo do dado armazenado naquela posição de memória. Assim, a declaração `float t1;`, do código mostrado, reserva um espaço de memória para armazenar um valor real (ponto flutuante – `float`). Esse espaço de memória é referenciado pelo símbolo `t1`.

Uma característica fundamental da linguagem C diz respeito ao tempo de vida e à visibilidade das variáveis. Uma variável (local) declarada dentro de uma função "vive" enquanto a função está sendo executada, e nenhuma outra função tem acesso direto a ela. Outra característica das variáveis locais é que devem sempre ser explicitamente inicializadas antes do uso, caso contrário carregarão "lixo", isto é, valores indefinidos.

Como alternativa, é possível definir variáveis externas às funções, ditas variáveis globais, que podem ser acessadas pelo nome por qualquer função subseqüente (são "visíveis" em todas as funções subseqüentes à sua definição). Além do mais, como as variáveis externas (ou globais) existem permanentemente (pelo menos enquanto o programa estiver sendo executado), elas retêm seus valores mesmo quando as funções que as acessam são finalizadas. Embora seja possível definir variáveis globais em qualquer parte do ambiente global (entre quaisquer funções), é prática comum defini-las no início do arquivo-fonte.

Como regra geral, por razões de clareza e estruturação adequada do código, devemos evitar o uso indisciplinado de variáveis globais e resolver os problemas por meio de variáveis locais sempre que possível. No próximo capítulo, discutiremos variáveis em mais detalhes.

Para desenvolver programas em uma linguagem como a C, precisamos de, no mínimo, um editor e um compilador. Esses programas têm finalidades bem definidas: com o editor de textos[1], escrevemos os programas-fontes, salvos em arquivos; com o compilador, transformamos os programas-fontes em programas objetos (linguagem de máquina), conforme discutimos na seção anterior. Os programas-fontes são, em geral, armazenados em arquivos cujo nome tem a extensão ".c". Os programas executáveis possuem extensões que variam dependendo do sistema operacional: no Windows, têm extensão ".exe"; no Unix (Linux), em geral, não têm extensão.

Consideremos que o código apresentado anteriormente foi compilado e gerou o executável correspondente. Se executarmos esse programa, teremos:

```
Digite a temperatura em Celsius: 10
A temperatura em Fahrenheit vale: 50.000000
```

Em itálico, representamos as mensagens do programa apresentadas na tela do computador e, em negrito, exemplificamos um dado fornecido pelo usuário via teclado.

Ciclo de desenvolvimento

Programas como editores, compiladores e ligadores são às vezes chamados de "ferramentas", usados na construção de programas. Exceto no caso de programas muito pequenos (como em nosso exemplo), é raro que um programa seja composto de um único arquivo-fonte. Normalmente, para facilitar o projeto, os programas são divididos em vários arquivos. Cada um deles pode ser compilado em separado, mas, para a obtenção de um programa executável, é necessário reunir os códigos de todos eles, sem esquecer as bibliotecas necessárias – essa é a função do *ligador*.

A tarefa das bibliotecas é permitir que funções de interesse geral estejam disponíveis com facilidade. Nosso exemplo usa a biblioteca de entrada/saída padrão de C, *stdio*, que oferece funções para permitir a captura de dados a partir do teclado e a saída de dados para a tela, entre outras. Além de bibliotecas preparadas pelo fornecedor do compilador ou por outros fornecedores de software, podemos ter bibliotecas preparadas por um programador qualquer, que pode "empacotar" funções com utilidades relacionadas em uma biblioteca e, dessa maneira, facilitar seu uso em outros programas.

Em alguns casos, a função do ligador é executada pelo próprio compilador. Em geral, quando nosso programa é composto por um único programa-fonte, a

---

[1]Podemos utilizar qualquer editor de texto para escrever os programas-fontes, exceto editores que incluem caracteres de formatação (como o Microsoft® Word do Windows®, por exemplo).

etapa de compilação inclui automaticamente a etapa de ligação. De um modo simplificado, podemos pensar que o ligador, no nosso exemplo, foi responsável por reunir o código do programa escrito aos códigos de `scanf`, `printf` e de outras funções necessárias à execução independente do programa.

Verificação e validação

Outro ponto que deve ser observado é que os programas podem conter (e, com freqüência, contêm) erros, que precisam ser identificados e corrigidos. Quase sempre a verificação é realizada por meio de testes, que executam o programa a ser testado com diferentes valores de entrada. Identificado um ou mais erros, o código-fonte é corrigido e deve ser novamente verificado. O processo de edição, compilação, ligação e teste é repetido até que os resultados sejam satisfatórios, e o programa seja considerado validado. A Figura 1.3 ilustra o ciclo de desenvolvimento de programas.

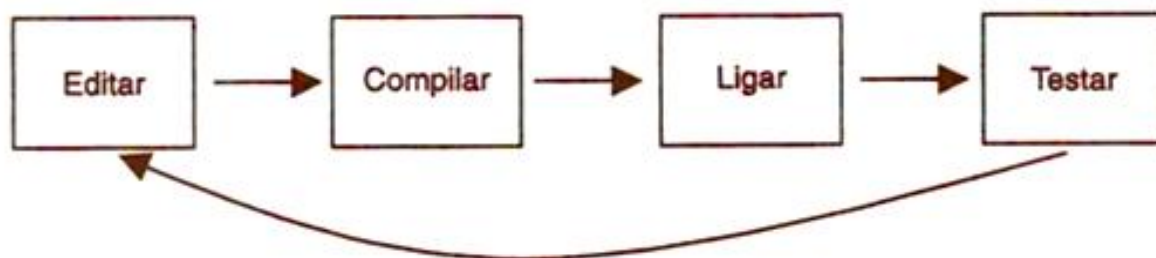



Figura 1.3 *Ciclo de desenvolvimento.*

Esse ciclo pode ser realizado com programas (editor, compilador, ligador) separados ou em um "ambiente integrado de desenvolvimento" (*Integrated Development Environment*, ou IDE). O IDE é um programa que oferece janelas para a edição de programas e facilidades para abrir, fechar e salvar arquivos e para compilar, ligar e executar programas. Se um IDE estiver disponível, é possível criar e testar um programa, tudo em um mesmo ambiente, e todo o ciclo mencionado acontece de maneira mais confortável dentro de um mesmo ambiente, de preferência com uma interface amigável.

2

Expressões

Na linguagem de programação C, uma expressão é uma combinação de variáveis, constantes e operadores que pode ser avaliada computacionalmente, resultando em um valor. O valor resultante é chamado de valor da expressão.

Variáveis

Podemos dizer que uma variável representa um espaço na memória do computador para armazenar um determinado tipo de dado. Na linguagem C, todas as variáveis devem ser explicitamente declaradas. Na declaração de uma variável, devem ser especificados seu *tipo* e seu *nome*: o nome da variável serve de referência ao dado armazenado no espaço de memória da variável e o tipo da variável determina a natureza do dado que será armazenado. Só podemos armazenar valores do tipo especificado na declaração da variável. Assim, se declararmos uma variável como sendo do tipo inteiro, só podemos armazenar valores inteiros no espaço de memória correspondente.

Tipos básicos

A linguagem C oferece alguns tipos básicos. Para armazenar valores inteiros, existem quatro tipos básicos: `char`, `short int`, `int`, `long int`. Esses tipos diferem entre si pelo espaço de memória que ocupam e, conseqüentemente, pelo intervalo de valores que podem representar. O tipo `char`, por exemplo, ocupa 1 byte de memória (8 bits), e pode representar 2^8 (=256) valores distintos. Os tipos `short int` e `long int` podem ser referenciados simplesmente como `short` e `long`, respectivamente. Na maioria das implementações da linguagem C, o tipo `short` é representado por 2 bytes, e o tipo `long`, por 4 bytes. O tipo `int` puro é, em geral, mapeado para o tipo inteiro natural da máquina. Sua representatividade é maior

ou igual à do tipo `short` e menor ou igual à do tipo `long`. A maioria das máquinas usadas hoje funciona com processadores de 32 bits, e o tipo `int` é mapeado para o inteiro de 4 bytes (`long`).[1] Todos esses tipos podem ainda ser modificados para representar apenas valores positivos, o que pode ser feito precedendo o tipo com o modificador "sem sinal" – `unsigned`. A Tabela 2.1 compara os tipos para valores inteiros e suas representatividades.

Tabela 2.1 Tipos de valores inteiros e suas representatividades

Tipo	Tamanho	Representatividade
char	1 byte	–128 a 127
unsigned char	1 byte	0 a 255
short int	2 bytes	–32 768 a 32 767
unsigned short int	2 bytes	0 a 65 535
long int	4 bytes	–2 147 483 648 a 2 147 483 647
unsigned long int	4 bytes	0 a 4 294 967295

O tipo `char` costuma ser usado apenas para representar códigos de caracteres, como veremos nos capítulos subseqüentes. Na prática, salvo situações específicas, usamos o tipo `int`, sem modificadores, para representar números inteiros.

A linguagem oferece ainda dois tipos básicos para a representação de números reais (ponto flutuante): `float` e `double`. O tipo `double` (precisão dupla) é recomendado para as situações nas quais a precisão numérica das operações é de fundamental importância. Por exemplo, em aplicações que fazem simulações numéricas, em geral precisamos trabalhar com maior precisão. A Tabela 2.2 compara esses dois tipos.

Tabela 2.2

Tipo	Tamanho	Representatividade
float	4 bytes	$\pm 10^{-38}$ a 10^{38}
double	8 bytes	$\pm 10^{-308}$ a 10^{308}

Declaração de variáveis

Para armazenar um dado (valor) na memória do computador, devemos reservar o espaço correspondente ao tipo do dado. A declaração de uma variável reserva

[1] Um contra-exemplo é o compilador TurboC, desenvolvido para o sistema operacional DOS, mas que ainda pode ser utilizado no sistema operacional Windows®. No TurboC, o tipo `int` é mapeado para 2 bytes.

um espaço na memória para armazenar um dado do tipo da variável e associa o nome da variável a esse espaço de memória.

Por exemplo, no fragmento de código a seguir, declaramos duas variáveis, a e b, para armazenar valores inteiros (tipo int) e uma variável, c, para armazenar valores reais (tipo float). Uma vez declaradas as variáveis, podemos armazenar valores dos tipos correspondentes. Isso é feito atribuindo-se valores às variáveis.

```
int a;          /* declara uma variável do tipo int */
int b;          /* declara outra variável do tipo int */
float c;        /* declara uma variável do tipo float */

a = 5;          /* armazena o valor 5 em a */
b = 10;         /* armazena o valor 10 em b */
c = 5.3;        /* armazena o valor 5.3 em c */
```

A linguagem permite que variáveis de mesmo tipo sejam declaradas juntas. Assim, essas duas primeiras declarações poderiam ser substituídas por:

```
int a, b;       /* declara duas variáveis do tipo int */
```

Uma vez declarada a variável, só podemos armazenar valores do mesmo tipo da variável, conforme ilustrado aqui. Não é possível, por exemplo, armazenar um número real numa variável do tipo int. Se fizermos:

```
int a;
a = 4.3;        /* a variável armazenará o valor 4 */
```

será armazenada em a apenas a parte inteira do número real, isto é, 4. Alguns compiladores exibem uma advertência quando encontram esse tipo de atribuição.

Em C, as variáveis podem ser inicializadas na declaração. Podemos, por exemplo, escrever:

```
int a = 5, b = 10;   /* declara e inicializa as variáveis */
float c = 5.3;
```

Valores constantes

Em nossos códigos, usamos também valores constantes. Quando escrevemos a atribuição

```
a = b + 123;
```

sendo a e b variáveis supostamente já declaradas, deve-se representar internamente também a constante 123, para que a operação possa ser avaliada em tempo

de execução. Podemos dizer que esse valor constante está armazenado em um espaço de memória próprio. No caso, a constante é do tipo inteiro, então um espaço de quatro bytes (em geral) seria reservado, e o valor 123 armazenado nele. A diferença básica em relação às variáveis, como os nomes dizem (variáveis e constantes), é que o valor armazenado em uma área de constante não pode ser alterado.

As constantes também podem ser do tipo real. Uma constante real deve ser escrita com um ponto decimal ou valor de expoente. Sem nenhum sufixo, uma constante real é do tipo `double`. Se quisermos uma constante real do tipo `float`, devemos, a rigor, acrescentar o sufixo `F` ou `f`. Alguns exemplos de constantes reais são:

```
12.45      constante real do tipo double
1245e-2    constante real do tipo double
12.45F     constante real do tipo float
```

Alguns compiladores exibem uma advertência quando encontram este código:

```
float x;
...
x = 12.45;
```

pois o código, a rigor, armazena um valor `double` (`12.45`) em uma variável do tipo `float`. Desde que a constante seja representável dentro de um `float`, não precisamos nos preocupar com esse tipo de advertência. Todavia, se quisermos evitá-lo, podemos representar a constante em precisão `float`:

```
float x;
...
x = 12.45f;
```

Variáveis com valores indefinidos

Um dos erros comuns em programas de computador é o uso de variáveis cujos valores ainda estão indefinidos. Se declaramos uma variável sem explicitamente inicializar seu valor, ele é indefinido. Existe um valor armazenado, representado pela seqüência de bits do espaço reservado, mas, como não temos controle sobre esse valor, não faz sentido utilizá-lo. Costumamos dizer que o valor da variável é "lixo". Por exemplo, o trecho de código a seguir está errado, pois o valor armazenado na variável `b` está indefinido e tentamos usá-lo na atribuição a `c`.

```
int a, b, c;
a = 2;
c = a + b;            /* ERRO: b tem "lixo" */
```

Alguns desses erros são óbvios (como o ilustrado), e o compilador é capaz de nos reportar uma advertência. No entanto, muitas vezes o uso de uma variável não definida é difícil de ser identificado no código. É importante ressaltar que esse é um erro comum em programas, e é uma razão para alguns programas funcionarem na parte da manhã e não funcionarem na parte da tarde (ou funcionarem durante o desenvolvimento e não funcionarem quando os entregamos ao cliente!). Todos os erros em computação têm lógica. A razão de o programa funcionar uma vez e não funcionar outra é que, como já mencionamos, apesar de indefinido, o valor da variável existe. No nosso caso citado anteriormente, pode acontecer de o valor armazenado na memória ocupada por `b` ser `0`, fazendo com que o programa funcione. Por outro lado, pode acontecer de o valor ser, por exemplo, `-293423` e o programa não funcionar conforme esperado.

Operadores

A linguagem C oferece uma gama variada de operadores, entre binários e unários. Os operadores binários operam sobre dois operandos, enquanto os operadores unários operam sobre um operando. Em C, um operador binário é escrito entre seus dois operandos, e um operador unário precede seu único operando. Os operadores básicos da linguagem são apresentados a seguir.

Operadores aritméticos

Os operadores aritméticos binários são: adição (+), subtração (-), multiplicação (*), divisão (/) e o operador módulo (%). Há ainda o operador menos unário (-). A operação é feita na precisão dos operandos. Assim, a expressão `5/2` resulta no valor `2`, pois a operação de divisão é feita em precisão inteira, já que os dois operandos (`5` e `2`) são constantes inteiras. A divisão de inteiros trunca a parte fracionária, pois o valor resultante é sempre do mesmo tipo da expressão. Conseqüentemente, a expressão `5.0/2.0` resulta no valor real `2.5` pois a operação é feita na precisão real (`double`, no caso).

Como as operações são feitas na precisão dos operandos, devemos ser cautelosos quando codificamos expressões, para que o resultado obtido seja o esperado. Para exemplificar, vamos considerar este fragmento de código:

```
int a;
double b, c;
...
a = 3.5;
b = a / 2.0;
c = 1/3 + b;
```

Se executado, esse fragmento de código armazenará os valores `3`, `1.5` e `1.5` nas variáveis `a`, `b` e `c`, respectivamente. Na primeira atribuição (`a = 3.5;`), o valor

armazenado em a é 3, isto é, o valor 3.5 é implicitamente convertido em inteiro e armazenado em a, uma vez que a só pode armazenar valores inteiros. Na segunda atribuição (b = a/2.0;), o valor de a (int, igual a 3) é dividido pelo valor 2.0 (double). Nesse caso, como os operandos são de tipos distintos, o valor do operando de menor expressividade (no caso, int) é implicitamente convertido para o tipo de maior expressividade (double), e a operação é feita na precisão double, o que resulta no valor 1.5, armazenado em b. A seguir, em c, também é armazenado o valor 1.5 pois a subexpressão 1/3 (divisão na precisão inteira) resulta em zero.

O operador módulo, %, não se aplica a valores reais e seus operandos devem ser do tipo inteiro. Ele produz o resto da divisão do primeiro pelo segundo operando. Como exemplo de aplicação desse operador, podemos citar o caso em que desejamos saber se o valor armazenado numa determinada variável inteira x é par ou ímpar. Para tanto, basta analisar o resultado da aplicação do operador %, aplicado à variável e ao valor dois.

```
x % 2   se resultado for zero   ⇒   número é par
x % 2   se resultado for um     ⇒   número é ímpar
```

Os operadores *, / e % têm precedência maior do que os operadores + e -. O operador - unário tem precedência maior do que *, / e %. Operadores com a mesma precedência são avaliados da esquerda para a direita. Assim, na expressão

```
a + b * c /d
```

executa-se primeiro a multiplicação, seguida da divisão e da soma. Podemos utilizar parênteses para alterar a ordem de avaliação de uma expressão. Assim, se quisermos avaliar a soma primeiro, podemos escrever:

```
(a + b) * c /d
```

É apresentada uma tabela de precedência dos operadores da linguagem C no final desta seção.

Operadores de atribuição

Na linguagem C, uma atribuição é uma expressão cujo valor resultante corresponde ao valor atribuído. Assim, da mesma forma que a expressão

```
5 + 3
```

resulta no valor 8, a atribuição

```
a = 5
```

é uma expressão que resulta no valor 5 (além, é claro, de armazenar o valor 5 na variável a). Esse tratamento das atribuições permite escrever comandos do tipo:

```
y = x = 5;
```

Nesse caso, a ordem de avaliação é da direita para a esquerda. Assim, o computador avalia x = 5, armazenando 5 em x, e, em seguida, armazena em y o valor produzido por x = 5, que é 5. Portanto, x e y recebem o valor 5.

A linguagem também permite utilizar os chamados operadores de atribuição compostos. Comandos do tipo:

```
i = i + 2;
```

em que a variável à esquerda do sinal de atribuição também aparece à direita podem ser escritos de forma mais compacta:

```
i += 2;
```

usando o operador de atribuição composto +=. Analogamente, existem, entre outros, os operadores de atribuição: -=, *=, /=, %=. De forma geral, comandos do tipo:

```
var op= expr;
```

são equivalentes a:

```
var = var op (expr);
```

Salientamos a presença dos parênteses em torno de expr. Assim:

```
x *= y + 1;
```

equivale a

```
x = x * (y + 1)
```

e não a

```
x = x * y + 1;
```

Operadores de incremento e decremento

A linguagem C apresenta ainda dois operadores não convencionais. São os operadores de incremento e decremento, que possuem precedência comparada ao -

unário e servem para incrementar e decrementar uma unidade nos valores armazenados nas variáveis. Assim, se n é uma variável que armazena um valor, o comando:

```
n++;
```

incrementa em uma unidade o valor de n (análogo para o decremento em n--). O aspecto não usual é que ++ e -- podem ser usados como operadores prefixados (antes da variável, como em ++n) ou pós-fixados (após a variável, como em n++). Em ambos os casos, a variável n é incrementada. Entretanto, a expressão ++n incrementa n antes de usar seu valor, enquanto n++ incrementa n após o valor ser usado. Isso significa que, em um contexto em que o valor de n é usado, ++n e n++ são diferentes. Se n armazena o valor 5, então:

```
x = n++;
```

atribui 5 a x, mas

```
x = ++n;
```

atribuiria 6 a x. Em ambos os casos, n passa a valer 6, pois seu valor foi incrementado em uma unidade. Analogamente, o fragmento de código:

```
a = 3;
b = a++ * 2;
```

resultaria no armazenamento dos valores 4 e 6 nas variáveis a e b, respectivamente. Os operadores de incremento e decremento podem ser aplicados somente em variáveis; uma expressão do tipo x = (i + 1)++ é ilegal.

A linguagem C oferece diversas formas compactas para escrever um determinado comando. Nos nossos exemplos, procuraremos evitar as formas compactas pois elas tendem a dificultar a compreensão do código. Mesmo para programadores experientes, o uso das formas compactas deve ser feito com critério. Por exemplo, os comandos:

```
a = a + 1;
a += 1;
a++;
++a;
```

são todos equivalentes, e o programador deve escolher o que achar mais adequado e simples. Em termos de desempenho, qualquer compilador razoável é capaz de otimizar todos esses comandos da mesma forma.

Operadores relacionais e lógicos

Os operadores relacionais são usados para comparar dois valores. A linguagem C oferece os seguintes operadores relacionais:

`<` *menor que*
`>` *maior que*
`<=` *menor ou igual que*
`>=` *maior ou igual que*
`==` *igual a*
`!=` *diferente de*

Esses operadores comparam dois valores. O resultado produzido por um operador relacional é zero ou um. Em C, não existe o tipo booleano (*true* ou *false*). O valor zero é interpretado como falso, e qualquer valor diferente de zero é considerado verdadeiro. Assim, se o resultado de uma comparação for falso, produz-se o valor 0; caso contrário, produz-se o valor 1.

Os operadores lógicos servem para combinar expressões booleanas. A linguagem oferece os seguintes operadores lógicos:

`&&` *operador binário E (AND)*
`||` *operador binário OU (OR)*
`!` *operador unário de NEGAÇÃO (NOT)*

Expressões conectadas por `&&` ou `||` são avaliadas da esquerda para a direita, e a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida. Recomendamos o uso de parênteses em expressões que combinam operadores lógicos e relacionais.

Os operadores relacionais e lógicos são normalmente utilizados para codificar tomada de decisões, o que será discutido no próximo capítulo. No entanto, podemos utilizar esses operadores para atribuir valores a variáveis. Por exemplo, o trecho de código a seguir é válido e armazena o valor 1 em `a` e 0 em `b`.

```
int a, b;
int c = 23;
int d = c + 4;

a = (c < 20) || (d > c);    /* verdadeiro */
b = (c < 20) && (d > c);    /* falso */
```

Devemos salientar que, na avaliação da expressão atribuída à variável `b`, a operação `(d>c)` não chega a ser avaliada, pois, independente do resultado, a expressão terá como resultado `0` (falso), uma vez que a operação `(c<20)` tem valor falso.

Operador `sizeof`

Outro operador fornecido por C, `sizeof`, resulta no número de bytes de um determinado tipo. Por exemplo:

```
int a = sizeof(float);
```

armazena o valor 4 na variável a, pois um `float` ocupa 4 bytes de memória. Esse operador pode também ser aplicado a uma variável, retornando o número de bytes ocupado pela variável.

Conversão de tipo

Em C, como na maioria das linguagens, existem conversões automáticas de valores na avaliação de uma expressão. Como já mencionamos, em uma expressão como `3/1.5`, o valor da constante 3 (tipo `int`) é promovido (convertido) para `double` antes de a expressão ser avaliada, pois o segundo operando é do tipo `double` (`1.5`), e a operação é feita na precisão do tipo mais representativo.

Quando, em uma atribuição, o tipo do valor atribuído é diferente do tipo da variável, também há uma conversão automática de tipo. Por exemplo, se escrevermos:

```
float a = 3;
```

o valor 3 é convertido para `float` (isto é, passa a valer `3.0F`) antes de a atribuição ser efetuada. Como resultado, conforme esperado, o valor atribuído à variável é `3.0F` (`float`). Alguns compiladores exibem advertências quando a conversão de tipo pode significar uma perda de precisão; é o caso quando armazenamos um número real em uma variável do tipo inteiro, ou quando armazenamos um `double` numa variável `float`.

O programador pode explicitamente requisitar uma conversão de tipo usando o operador de molde de tipo (operador *cast*). Por exemplo, são válidos (e isentos de qualquer advertência por parte dos compiladores) estes comandos:

```
int a, b;
a = (int) 3.5;
b = (int) 3.5 % 2;
```

Precedência e ordem de avaliação dos operadores

A Tabela 2.3 mostra a precedência, em ordem decrescente, dos principais operadores da linguagem C.

Tabela 2.3 Precedência (em ordem decrescente) e em ordem de avaliação dos operadores

Operador	Associatividade
() [] -> .	esquerda para direita
! ~ ++ -- - (tipo) * & sizeof(tipo)	direita para esquerda
* / %	esquerda para direita
+ -	esquerda para direita
<< >>	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
&	esquerda para direita
^	esquerda para direita
|	esquerda para direita
&&	esquerda para direita
||	esquerda para direita
?:	direita para esquerda
= += -= etc.	direita para esquerda
,	esquerda para direita

Entrada e saída básicas

A linguagem C não possui comandos de entrada e saída de dados. Tudo em C é feito com o uso de funções, inclusive as operações de entrada e saída. Por isso, já existe em C uma biblioteca padrão que possui as funções básicas normalmente necessárias. Nela, podemos, por exemplo, encontrar funções matemáticas do tipo raiz quadrada, seno, cosseno etc., funções para a manipulação de cadeias de caracteres e funções de entrada e saída. Nesta seção, serão apresentadas as duas funções básicas de entrada e saída disponibilizadas pela biblioteca padrão. Para utilizá-las, é necessário incluir o protótipo das funções no código. Esse assunto será tratado em detalhes na seção sobre funções. Por ora, basta saber que é preciso escrever:

```
#include <stdio.h>
```

no início do programa que utiliza as funções da biblioteca de entrada e saída.

Função `printf`

A função `printf` possibilita a saída de valores (constantes, variáveis ou resultados de expressões) segundo um determinado formato. Informalmente, podemos dizer que a forma da função é:

```
printf (formato, lista de constantes/variáveis/expressões...);
```

O primeiro parâmetro é uma cadeia de caracteres, em geral, delimitada com aspas, que especifica o formato de saída das constantes, variáveis e expressões listadas em seguida.

Para cada valor que se deseja imprimir, deve existir um especificador de formato correspondente na cadeia de caracteres *formato*. Os especificadores de formato variam com o tipo do valor e a precisão com que queremos que sejam impressos. Esses especificadores são precedidos pelo caractere % e podem ser, entre outros:

%c	*especifica um char*
%d	*especifica um int*
%u	*especifica um unsigned int*
%f	*especifica um double (ou float)*
%e	*especifica um double (ou float) no formato científico*
%g	*especifica um double (ou float) no formato mais apropriado (%f ou %e)*
%s	*especifica uma cadeia de caracteres*

Alguns exemplos:

```
printf ("%d %g\n", 33, 5.3);
```

tem como resultado a impressão da linha:

```
33 5.3
```

Ou:

```
printf ("Inteiro = %d   Real = %g\n", 33, 5.3);
```

com saída:

```
Inteiro = 33   Real = 5.3
```

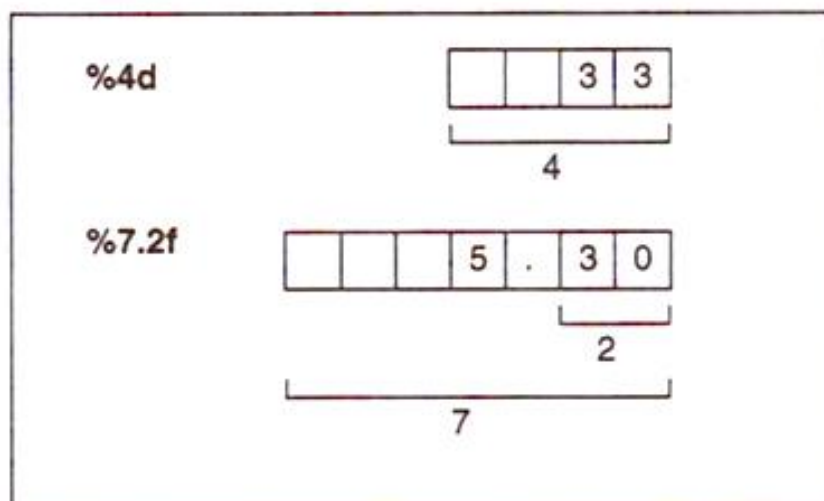
Isto é, além dos especificadores de formato, podemos incluir textos no formato, que são mapeados diretamente para a saída. Assim, a saída é formada pela cadeia de caracteres do formato em que os especificadores são substituídos pelos valores correspondentes. De fato, muitas vezes, queremos apenas exibir uma mensagem na tela e, nesses casos, o formato é o texto que será exibido, sem especificadores de formato. Por exemplo, o comando:

```
printf("Curso de Estruturas de Dados\n");
```

apenas exibe a mensagem Curso de Estruturas de Dados na tela. O caractere \n que aparece no final do formato apenas requisita uma mudança de linha. Assim, um eventual próximo printf exibiria a mensagem na linha seguinte. Além do caractere de nova linha, existem alguns outros caracteres de escape que são freqüentemente utilizados nos formatos de saída. São eles:

\n *caractere de nova linha*
\t *caractere de tabulação*
\r *caractere de retrocesso*
\" *o caractere "*
\\ *o caractere *

Ainda, se desejarmos ter na mensagem exibida um caractere %, devemos, dentro do formato, escrever %%.

É possível também especificar o tamanho dos campos:

Se, por exemplo, quisermos apenas fixar em 2 o número de casas decimais usadas para exibir valores reais, podemos usar o especificador de formato %.2f. Recomendamos a leitura do manual da linguagem C para uma discussão detalhada dos diversos especificadores de formato disponíveis.

Função scanf

A função scanf permite capturar valores fornecidos via teclado pelo usuário do programa e armazená-los em variáveis do nosso programa. Informalmente, podemos dizer que sua forma geral é:

```
scanf (formato, lista de endereços das variáveis...);
```

O formato deve possuir especificadores de tipos similares aos mostrados para a função `printf`. Para a função `scanf`, no entanto, existem especificadores diferentes para o tipo `float` e o tipo `double`:

`%c`	*especifica um char*
`%d`	*especifica um int*
`%u`	*especifica um unsigned int*
`%f,%e,%g`	*especificam um float*
`%lf, %le, %lg`	*especificam um double*
`%s`	*especifica uma cadeia de caracteres*

A principal diferença em relação à função `printf` é que o formato deve ser seguido por uma lista de endereços de variáveis (na função `printf` passamos os valores de constantes, variáveis e expressões). Na seção sobre ponteiros, esse assunto será tratado em detalhes. Por ora, basta saber que, para ler um valor e atribuí-lo a uma variável, devemos passar o endereço da variável para a função `scanf`. O operador `&` retorna o endereço de uma variável. Assim, para ler um inteiro, devemos ter:

```
int n;
scanf ("%d", &n);
```

Dessa forma, o valor inteiro digitado pelo usuário é armazenado na variável `n`.

Para a função `scanf`, os especificadores `%f`, `%e` e `%g` são equivalentes. Aqui, caracteres diferentes dos especificadores no formato servem para cercar a entrada. Por exemplo:

```
scanf ("%d:%d", &h, &m);
```

obriga que os valores (inteiros) fornecidos sejam separados pelo caractere dois-pontos (`:`). Um espaço em branco dentro do formato faz com que sejam "saltados" os eventuais brancos da entrada. Os especificadores `%d`, `%f`, `%e`, `%g` e `%s` pulam automaticamente os brancos que precederem os valores numéricos a serem capturados.

Para exemplificar o uso das funções de entrada e saída e a construção de expressões, vamos considerar um exemplo em que desejamos converter a altura de uma pessoa, dada em metros, para a altura expressa em pés e polegadas. Sabe-se que 1 pé tem 30,54 cm e que 1 polegada tem 2,54 cm. Assim, se o usuário entrar com o valor `1.8` (em metros), o programa deve exibir o valor `5ft 10.9pol`.

Um código que ilustra a implementação desse programa é mostrado a seguir.

```
/* Programa para converter altura em metros para ft e pol */

#include <stdio.h>

int main (void)
{
  int f;        /* número de pés */
  float p;      /* número de polegadas */
  float h;      /* altura em metros */

  /* Captura altura em metros */
  printf("Digite altura em metros: ");
  scanf("%f", &h);

  /* Calcula altura em pés e polegadas */
  h = 100*h;                /* converte para centímetros */
  f = (int) (h/30.48);      /* calcula número de pés */
  p = (h-f*30.48) / 2.54;   /* calcula número de polegadas do restante */

  /* Exibe altura convertida */
  printf("Altura: %dft %.1fpol\n", f, p);

  return 0;
}
```

3

Controle de fluxo

No capítulo anterior, trabalhamos com programas formados por seqüências simples de comandos. Para a construção de programas mais elaborados, precisamos ter acesso a mecanismos que permitam controlar o fluxo de execução dos comandos. Por exemplo, é fundamental ter meios para tomar decisões que se baseiem em condições avaliadas em tempo de execução. Também precisamos de mecanismos para a construção de procedimentos iterativos, isto é, procedimentos que repetem a execução de uma seqüência de comandos um determinado número de vezes.

Neste capítulo, discutiremos os principais mecanismos para controle de fluxo oferecidos pela linguagem C. Ela provê as construções fundamentais de controle de fluxo necessárias para programas bem estruturados: agrupamentos de comandos, tomadas de decisão (`if-else`), laços com teste de encerramento no início (`while`, `for`) ou no fim (`do-while`) e seleção de um caso entre um conjunto de casos possíveis (`switch`).

Tomada de decisão

O comando `if` é o comando básico para codificar tomada de decisão em C. Sua forma pode ser:

```
if (expr) {
  bloco de comandos 1
...
}
```

ou

```
if ( expr ) {
  bloco de comandos 1
...
}
else {
  bloco de comandos 2
...
}
```

Se a avaliação de *expr* resultar em um valor diferente de 0 (isto é, se o valor for verdadeiro), o bloco de comandos 1 será executado. A inclusão do `else` requisita a execução do bloco de comandos 2 se a expressão resultar no valor 0 (falso). Cada bloco de comandos deve ser delimitado por uma chave aberta e uma fechada. No entanto, se dentro de um bloco tivermos apenas um único comando a ser executado, as chaves podem ser omitidas (a rigor, deixamos de ter um bloco):

```
if ( expr )
  comando1;
else
  comando2;
```

As formas gerais do comando `if` apresentadas anteriormente ilustram o uso de códigos identados. A *identação* (recuo de linha) dos comandos não é obrigatória, mas é fundamental para uma maior clareza do código. Em geral, os blocos de comandos são identados (recuados) em relação ao comando `if` correspondente. Dessa forma, é fácil identificar visualmente o início e o fim de cada bloco. O estilo de identação varia a gosto do programador. Além da forma ilustrada, outro estilo bastante utilizado por programadores coloca a chave aberta na linha seguinte ao `if`:

```
if ( expr )
{
  bloco de comandos 1
...
}
else
{
  bloco de comandos 2
...
}
```

O número de espaços usados para a de blocos também varia a gosto do programador. O importante é ser consistente ao longo de todo o programa.

Para exemplificar o uso de comandos *if*, vamos considerar um programa que captura um valor inteiro fornecido via teclado e imprime uma mensagem infor-

mando se o número inserido é um número par ou ímpar. Um exemplo desse código simples é ilustrado a seguir:

```
#include <stdio.h>

int main (void)
{
  int a;
  printf("Digite um numero inteiro:");
  scanf("%d ",&a);
  if (a%2 == 0) {
     printf("O numero fornecido e' par!\n");
  }
  else {
     printf("O numero fornecido e' impar!\n");
  }
  return 0;
}
```

Podemos aninhar comandos `if`. Um exemplo simples pode ser:

```
#include <stdio.h>

int main (void)
{
  int a, b;
  printf("Digite dois numeros inteiros:");
  scanf("%d%d",&a,&b);
  if (a%2 == 0) {
     if (b%2 == 0) {
        printf("Foram digitados dois numeros pares!\n");
     }
  }
  return 0;
}
```

Para este último exemplo, devemos notar que a criação dos blocos ({...}) não era obrigatória, porque a cada `if` está associado apenas um único comando. Ao primeiro, associamos o segundo comando `if`, e ao segundo `if` associamos o comando que chama a função `printf`. Assim, o segundo `if` só será avaliado se o primeiro valor fornecido for par, e a mensagem só será impressa se o segundo valor fornecido também for par. Outra construção para esse mesmo exemplo pode ser:

```
int main (void)
{
  int a, b;
  printf("Digite dois numeros inteiros:");
  scanf("%d%d",&a,&b);
  if ((a%2 == 0) && (b%2 == 0)) {
     printf ( "Foram digitados dois numeros pares!\n");
  }
  return 0;
}
```

que produz resultados idênticos.

Novamente, o uso das chaves é opcional. Todavia, nem sempre é fácil identificar a associação de comandos sem os blocos. Para ilustrar essa discussão, consideremos o exemplo a seguir, que usa aninhamento de comandos `if-else`:

```
/* temperatura (versao 1 - incorreta) */
#include <stdio.h>

int main (void)
{
  int temp;
  printf("Digite a temperatura: ");
  scanf("%d", &temp);
  if (temp < 30)
     if (temp > 20)
        printf(" Temperatura agradavel \n");
  else
     printf(" Temperatura muito quente \n");
  return 0;
}
```

A idéia desse programa era imprimir a mensagem `Temperatura agradavel` se fosse fornecido um valor entre 20 e 30 e imprimir a mensagem `Temperatura muito quente` se fosse fornecido um valor maior do que 30. No entanto, vamos analisar o caso em que é fornecido o valor 5 para `temp`. Ao observar o código do programa, poderíamos pensar que nenhuma mensagem seria fornecida, pois o primeiro `if` daria resultado verdadeiro e então seria avaliado o segundo `if`. Nesse caso, teríamos um resultado falso e como, aparentemente, não há um comando `else` associado, nada seria impresso. Puro engano. A identação utilizada pode nos levar a erros de interpretação. O resultado para o valor 5 seria a mensagem `Temperatura muito quente`. Isto é, o programa está INCORRETO.

Em C, um `else` é associado ao último `if` que não tiver seu próprio `else`. Para os casos em que a associação entre `if` e `else` não está clara, recomendamos a criação explícita de blocos, mesmo contendo um único comando. Se reescrevermos o programa, podemos obter o efeito desejado.

```
/* temperatura  (versao 2) */
#include <stdio.h>

int main (void)
{
  int temp;
  printf ( "Digite a temperatura: " );
  scanf ( "%d", &temp );
  if ( temp < 30 )  {
     if ( temp > 20 )
        printf ( " Temperatura agradavel \n" );
  }
  else
     printf ( " Temperatura muito quente \n" );
  return 0;
}
```

Essa regra de associação do `else` propicia a construção do tipo `else-if` sem que se tenha o comando `elseif` explicitamente na gramática da linguagem. Na verdade, em C, construímos estruturas `else-if` com `if`s aninhados. Este exemplo é válido e funciona como esperado.

```
/* temperatura  (versao 3) */
#include <stdio.h>

int main (void)
{
  int temp;
  printf("Digite a temperatura: ");
  scanf("%d", &temp);

  if (temp < 10)
     printf("Temperatura muito fria \n");
  else if (temp < 20)
     printf(" Temperatura fria \n");
  else if (temp < 30)
     printf("Temperatura agradavel \n");
  else
     printf("Temperatura muito quente \n");
  return 0;
}
```

Estruturas de bloco

Observamos que uma função C é composta por estruturas de blocos. Cada chave aberta e fechada em C representa um bloco. As declarações de variáveis só po-

dem ocorrer no início do corpo da função ou no início de um bloco, isto é, devem seguir uma chave aberta[1]. Uma variável declarada dentro de um bloco é válida apenas dentro dele. Após o término do bloco, a variável deixa de existir. Por exemplo:

```
...
if ( n > 0 ) {
  int i;
  ...
}
...            /* a variável i não existe neste ponto do programa */
```

A variável `i`, definida dentro do bloco do `if`, só existe dentro deste bloco. É uma boa prática de programação declarar as variáveis o mais próximas possível de seus usos.

Operador condicional

C possui também um chamado operador condicional. Trata-se de um operador que substitui construções do tipo:

```
...
if ( a > b )
  maximo = a;
else
  maximo = b;
...
```

Sua forma geral é:

```
condição ? expressão1 : expressão2;
```

se a condição for verdadeira, a *expressão1* é avaliada; caso contrário, avalia-se a *expressão2*.

O comando:

```
maximo = a > b ? a : b ;
```

substitui a construção com `if-else` mostrada anteriormente.

[1] Esta limitação não existe no padrão C99 da linguagem C.

Construções com laços

Em programas computacionais, procedimentos iterativos são muito comuns, isto é, procedimentos que devem ser executados em vários passos. Como exemplo, vamos considerar o cálculo do valor do fatorial de um número inteiro não negativo. Por definição:

$$n! = n \times (n-1) \times (n-2) \ldots 3 \times 2 \times 1$$

onde: $0! = 1$

Para calcular o fatorial de um número com um programa de computador, normalmente utilizamos um processo iterativo, em que o valor da variável varia de 1 a n, avaliando o produtório.

A linguagem C oferece diversas construções possíveis para a realização de laços iterativos. O primeiro a ser apresentado é o comando `while`. Sua forma geral é:

```
while (expr) {
  bloco de comandos
  ...
}
```

Se a avaliação de *expr* resultar em verdadeiro, o bloco de comandos é executado. Ao final do bloco, a expressão volta a ser avaliada e, enquanto *expr* resultar em verdadeiro, o bloco de comandos é executado repetidamente. Quando *expr* for avaliada em falso, o bloco de comando deixa se ser executado, e a execução do programa prossegue com a execução dos comandos subseqüentes ao bloco. Uma possível implementação do cálculo do fatorial usando `while` é mostrada a seguir.

```
/* Fatorial */

#include <stdio.h>

int main (void)
{
  int i;
  int n;
  int f = 1;

  printf("Digite um número inteiro nao negativo:");
  scanf("%d", &n);
```

```
  /* calcula fatorial */
  i = 1;
  while (i <= n) {
     f *= i;
     i++;
  }

  printf(" Fatorial = %d \n", f);
  return 0;
}
```

Uma segunda forma de construção de laços em C, mais compacta e amplamente utilizada, é com laços for. Sua forma geral é:

```
for (expr_inicial; expr_booleana; expr_de_incremento) {
  bloco de comandos
  ...
}
```

A construção com for é equivalente ao uso do while, com a ordem de avaliação das expressões ilustradas a seguir:

```
expr_inicial;
while (expr_booleana) {
  bloco de comandos
  ...
  expr_de_incremento
}
```

Isto é, a expressão inicial é avaliada uma única vez antes da execução do laço. Em seguida, a expressão booleana, que controla a execução do laço, é avaliada e, enquanto for verdadeira, o bloco de comandos é executado. Imediatamente após cada execução do bloco de comandos, a expressão de incremento é avaliada, o laço se completa e a expressão booleana volta a ser avaliada.

A seguir, ilustramos a utilização do comando for no programa para cálculo do fatorial.

```
/* Fatorial (versao 2) */

#include <stdio.h>

int main (void)
{
  int i;
  int n;
  int f = 1;
```

```
  printf("Digite um número inteiro nao negativo:");
  scanf("%d", &n);

  /* calcula fatorial */
  for (i = 1; i <= n; i++) {
     f *= i;
  }
  printf(" Fatorial = %d \n", f);
  return 0;
}
```

Observamos que as chaves que seguem o comando `for`, nesse caso, são desnecessárias, já que o corpo do bloco é composto por um único comando.

Tanto a construção com `while` como a construção com `for` avaliam a expressão booleana que caracteriza o teste de encerramento no início do laço. Assim, se essa expressão tiver valor igual a zero (falso), quando for avaliada pela primeira vez, os comandos do corpo do bloco não serão executados nem uma vez.

C provê outro comando para a construção de laços cujo teste de encerramento é avaliado no final. Essa construção é o `do-while`, cuja forma geral é:

```
do
{
  bloco de comandos
  ...
} while (expr_booleana);
```

Um exemplo do uso dessa construção é mostrado a seguir, em que validamos a inserção do usuário, isto é, o programa repetidamente requisita a inserção de um número enquanto o usuário inserir um inteiro negativo (cujo fatorial não está definido).

```
/* Fatorial (versao 3) */

#include <stdio.h>

int main (void)
{
  int i;
  int n;
  int f = 1;

  /* requisita valor do usuário */
  do {
     printf("Digite um valor inteiro nao negativo:");
     scanf ("%d", &n);
  } while (n<0);
```

```
    /* calcula fatorial */
    for (i = 1; i <= n; i++)
        f *= i;

    printf(" Fatorial = %d\n", f);
    return 0;
}
```

Interrupções com `break` *e* `continue`

A linguagem C oferece ainda duas formas para a interrupção antecipada de um determinado laço. O comando `break`, quando utilizado dentro de um laço, interrompe e termina a execução do mesmo. A execução prossegue com os comandos subseqüentes ao bloco. O código a seguir ilustra o efeito de sua utilização.

```
#include <stdio.h>

int main (void)
{
    int i;
    for (i = 0; i < 10; i++) {
        if (i == 5)
            break;
        printf("%d  ", i);
    }
    printf("fim\n");
    return 0;
}
```

A saída desse programa, se executado, será:

```
0 1 2 3 4  fim
```

pois, quando `i` tiver o valor 5, o laço será interrompido e finalizado pelo comando `break`, passando o controle para o próximo comando após o laço, no caso uma chamada final de `printf`.

O comando `continue` também interrompe a execução dos comandos de um laço. A diferença básica em relação ao comando `break` é que o laço não é automaticamente finalizado. O comando `continue` interrompe a execução de um laço e passa para a próxima iteração. Assim, o código:

```
#include <stdio.h>

int main (void)
{
  int i;
  for (i = 0; i < 10; i++ ) {
     if (i == 5) continue;
     printf("%d  ", i);
  }
  printf("fim\n");
  return 0;
}
```

gera a saída:

```
0  1  2  3  4  6  7  8  9  fim
```

Devemos ter cuidado com a utilização do comando `continue` nos laços `while`. O programa:

```
/* INCORRETO */

#include <stdio.h>

int main (void)
{
  int i = 0;
  while (i < 10) {
     if (i == 5) continue;
     printf("%d  ", i);
     i++;
  }
  printf("fim\n");
  return 0;
}
```

é um programa INCORRETO, pois o laço criado não tem fim; a execução do programa não termina. Isto porque a variável `i` nunca terá valor superior a 5, e o teste será sempre verdadeiro. O que ocorre é que o comando `continue` "pula" os demais comandos do laço quando `i` vale 5, inclusive o comando que incrementa a variável `i`.

Seleção

Além da construção `else-if`, C provê um comando (`switch`) para selecionar um entre um conjunto de casos possíveis. Sua forma geral é:

```
switch ( expr ) {
  case op1:
     ...      /* comandos executados se expr == op1  */
  break;
  case op2:
     ...      /* comandos executados se expr == op2  */
  break;
  case op3:
     ...      /* comandos executados se expr == op3  */
  break;
  default:
     ...      /* executados se expr for diferente de todos  */
  break;
}
```

op_i deve ser um número inteiro ou uma constante caractere. Se *expr* resultar no valor op_i, os comandos seguintes ao caso op_i são executados até encontrar um break. Se o comando break for omitido, a execução do caso continua com os comandos do caso seguinte. Se valor de *expr* for diferente de todos os casos enumerados, o bloco de comandos associado a default é executado. O bloco default pode aparecer em qualquer posição, mas normalmente é colocado por último (pode também ser omitido).

Para exemplificar, mostramos a seguir um programa responsável por implementar uma calculadora convencional que efetua as quatro operações básicas. Esse programa usa constantes caracteres, que serão discutidas em detalhe quando apresentarmos cadeias de caracteres em C. O importante aqui é entender conceitualmente a construção switch.

```
/* calculadora de quatro operações */

#include <stdio.h>

int main (void)
{
  float num1, num2;
  char op;

  printf("Digite: numero op numero\n");
  scanf ("%f %c %f", &num1, &op, &num2);
  switch (op) {
     case '+':
        printf(" = %f\n", num1+num2);
     break;
     case '-':
        printf(" = %f\n", num1-num2);
     break;
```

```
      case '*':
         printf(" = %f\n", num1*num2);
      break;
      case '/':
         printf(" = %f\n", num1/num2);
      break;
      default:
         printf("Operador invalido!\n");
      break;
   }
   return 0;
}
```

4

Funções

Para a construção de programas estruturados, é sempre preferível dividir as grandes tarefas de computação em tarefas menores e utilizar seus resultados parciais para compor o resultado final desejado. Na linguagem C, a criação de funções é o mecanismo adequado para codificar tarefas específicas. Um programa estruturado em C deve ser composto por diversas funções pequenas. Essa estratégia de codificação traz dois grandes benefícios: primeiro, facilita a codificação, pois codificar diversas funções pequenas, que resolvem problemas específicos, é mais fácil do que codificar uma única função maior; segundo, funções específicas podem ser facilmente reutilizadas em outros códigos. De fato, a criação de funções pode evitar a repetição de código, de modo que um procedimento repetido deve ser transformado em uma função que, então, será chamada diversas vezes. Um programa deve ser pensado em termos de funções, que, por sua vez, podem (e devem, se possível) esconder do corpo principal do programa detalhes ou particularidades de implementação. Em C, tudo é feito usando funções. Os exemplos anteriores utilizam as funções da biblioteca padrão para realizar entrada e saída. Neste capítulo, discutiremos a codificação de nossas próprias funções.

Definição de funções

A forma geral para definir uma função é:

```
tipo_retornado  nome_da_função  ( lista de parâmetros... )
{
  corpo da função
}
```

Para ilustrar a criação de funções, consideraremos novamente o cálculo do fatorial de um número inteiro. Podemos escrever uma função que, dado um de-

terminado número inteiro não negativo n, imprime o valor de seu fatorial. Um programa que utiliza essa função seria:

```
/* programa que lê um número e imprime seu fatorial */

#include <stdio.h>

void fat (int n);

/* Função principal */
int main (void)
{
  int n;
  scanf("%d", &n);
  fat(n);
  return 0;
}

/* Função para imprimir o valor do fatorial */
void fat ( int n )
{
  int i;
  int f = 1;
  for (i = 1; i <= n; i++)
     f *= i;
  printf("Fatorial = %d\n", f);
}
```

Notamos, nesse exemplo, que a função fat recebe como parâmetro o número cujo fatorial deve ser impresso. Os parâmetros de uma função devem ser listados, com os respectivos tipos, entre os parênteses que seguem o nome da função. Quando uma função não tem parâmetros, colocamos a palavra reservada void entre parênteses. Devemos notar que main também é uma função; sua única particularidade consiste em ser a função automaticamente executada após o programa ser carregado. Como as funções main apresentadas até então não recebem parâmetros, usamos a palavra void na lista de parâmetros.

Além de receber parâmetros, uma função pode ter um valor de retorno associado. No exemplo do cálculo do fatorial, a função fat não tem nenhum valor de retorno, portanto colocamos a palavra void antes do nome da função, indicando a ausência de um valor de retorno.

```
void fat (int n)
{
  . . .
}
```

A função `main` deve ter obrigatoriamente um valor inteiro como retorno. Esse valor pode ser usado pelo sistema operacional para testar a execução do programa. A convenção geralmente utilizada faz com que a função `main` retorne zero, no caso de a execução ser bem-sucedida, ou diferente de zero, no caso de problemas durante a execução.

Por fim, salientamos que C exige que se coloque o protótipo da função antes de ela ser chamada. O protótipo de uma função consiste na repetição da linha de sua definição seguida do caractere (;). Temos então:

```
void fat (int n);      /* obs: existe ; no protótipo */

int main (void)
{
  . . .
}

void fat (int n)       /* obs: não existe ; na definição */
{
  . . .
}
```

A rigor, no protótipo não há necessidade de indicar os nomes dos parâmetros, apenas os tipos; portanto, seria válido escrever: `void fat (int);`. Entretanto, geralmente mantemos os nomes dos parâmetros, pois servem como documentação do significado de cada parâmetro, desde que sejam utilizados nomes coerentes. O protótipo da função é necessário para que o compilador verifique os tipos dos parâmetros na chamada da função. Por exemplo, se tentássemos chamar a função com `fat(4.5);` o compilador provavelmente indicaria o erro, pois estaríamos passando um valor real enquanto a função espera um valor inteiro. Por isso, exige-se a inclusão do arquivo *stdio.h* para a utilização das funções de entrada e saída da biblioteca padrão. Nesse arquivo, encontram-se, entre outras coisas, os protótipos das funções `printf` e `scanf`.

Uma função pode ter um valor de retorno associado. Para ilustrar a discussão, vamos reescrever o código anterior, fazendo com que a função `fat` retorne o valor do fatorial. A função `main` fica então responsável pela impressão do valor.

```
/* programa que lê um número e imprime seu fatorial (versão 2) */
#include <stdio.h>
int fat (int n);
int main (void)
{
  int n, r;
  scanf("%d", &n);
  r = fat(n);
  printf("Fatorial = %d\n", r);
  return 0;
}
```

```
/* função para calcular o valor do fatorial */
int fat (int n)
{
  int i;
  int f = 1;
  for (i = 1; i <= n; i++)
     f *= i;
  return f;
}
```

De fato, essa segunda implementação da função fat é mais adequada, pois a tarefa executada pela função se limita a fazer o cálculo do fatorial. A decisão de imprimir ou não o resultado na tela deve ficar a cargo da função que chama a função – denominada função cliente. Dessa forma, a função fat pode ser mais facilmente reutilizada. Por exemplo, podemos usar a função fat para avaliar qualquer expressão que envolva o cálculo do fatorial. Para ilustrar, consideremos o cálculo do número de combinações de n elementos tomados k a k, no qual a ordem dos elementos é relevante. Esse número é dado pela fórmula do arranjo:

$$a = \frac{n!}{(n-k)!}$$

Por meio da função fat, podemos facilmente implementar uma função para o cálculo do número de arranjos:

```
int arranjo (int n, int k)
{
  int a;
  a = fat(n) / fat(n-k);
  return a;
}
```

Ou simplesmente:

```
int arranjo (int n, int k)
{
  return fat(n) / fat(n-k);
}
```

Pilha de execução

Uma vez apresentada a forma básica para a definição de funções, discutiremos agora, em detalhes, como funciona a comunicação entre a função que chama e

a função chamada. As funções são independentes entre si. As variáveis locais definidas dentro do corpo de uma função (incluídos os parâmetros das funções) não existem fora dela. Cada vez que a função é executada, as variáveis locais são criadas, e, quando sua execução termina, as variáveis deixam de existir.

A transferência de dados entre funções é feita com o uso de parâmetros e do valor de retorno da função chamada. Conforme mencionado, uma função pode retornar um valor para a função que a chamou, o que é feito com o comando return. Quando uma função tem um valor de retorno, sua chamada é uma expressão cujo valor resultante é o valor retornado pela função. Por isso, foi válido escrever as expressões `r = fat(n) ;` e `a = fat(n)/fat(n-k)` , que chamam a função `fat` e usam o valor de retorno dentro de uma expressão (como o resultado atribuído a uma variável ou como operando de uma operação de divisão).

A comunicação por meio de parâmetros requer uma análise mais detalhada. Para ilustrar a discussão, vamos considerar o exemplo a seguir, no qual a implementação da função `fat` foi ligeiramente alterada:

```
/* programa que lê um numero e imprime seu fatorial (versão 3) */

#include <stdio.h>

int fat (int n);

int main (void)
{
  int n = 5;
  int r;
  r = fat ( n );
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}

int fat (int n)
{
  int f = 1;
  while (n != 0) {
     f *= n;
     n--;
  }
  return f;
}
```

Nesse exemplo, podemos verificar que, no final da função `fat`, o parâmetro `n` tem valor igual a zero (essa é a condição de encerramento do laço `while`). No entanto, a saída do programa será:

```
Fatorial de 5 = 120
```

pois o valor da variável `n` não mudou no programa principal, porque a linguagem C trabalha com o conceito de passagem por valor. Na chamada de uma função, o valor passado é atribuído ao parâmetro da função chamada. Cada parâmetro funciona como uma variável local inicializada com o valor passado na chamada. Assim, a variável `n` (parâmetro da função `fat`) é local e não representa a variável `n` da função `main` (o fato de as duas variáveis terem o mesmo nome é indiferente; poderíamos chamar o parâmetro de `v`, por exemplo). Alterar o valor de `n` dentro de `fat` não afeta o valor da variável `n` de `main`.

A execução do programa funciona com o modelo de pilha. De forma simplificada, o modelo de pilha funciona assim: cada variável local de uma função é colocada na pilha de execução. Ao chamar uma função, os parâmetros são copiados para a pilha e tratados como se fossem variáveis locais da função chamada. Quando a função termina, a parte da pilha correspondente àquela função é liberada, e, por isso, não podemos acessar as variáveis locais de fora da função em que foram definidas.

Para exemplificar, vamos considerar um esquema representativo da memória do computador – salientando que esse esquema é apenas uma maneira didática de explicar o que ocorre na memória do computador. Suponhamos que as variáveis são armazenadas na memória como ilustrado a seguir. Os números à direita representam endereços (posições) fictícios de memória, e os nomes à esquerda indicam os nomes das variáveis. A Figura 4.1 ilustra esse esquema representativo da memória que adotaremos.

Figura 4.1 *Esquema representativo da memória.*

Podemos, então, analisar passo a passo a evolução do programa mostrado anteriormente e, assim, ilustrar o funcionamento da pilha de execução.

A Figura 4.2 ilustra por que o valor da variável passada nunca será alterado dentro da função. A seguir, discutiremos uma forma de alterar valores por passagem de parâmetros, o que será realizado pela passagem do endereço de memória em que a variável está armazenada.

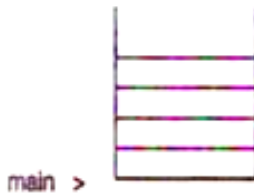
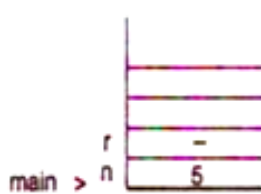
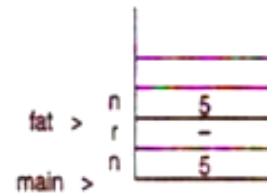
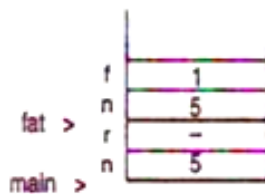
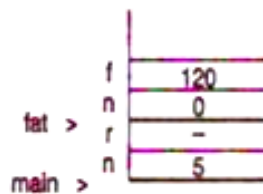
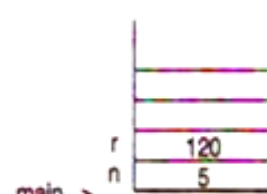



Figura 4.2 *Execução do programa passo a passo.*

Ponteiro de variáveis

Conforme ilustrado, uma função pode retornar um tipo de valor para a função que chama. Algumas versões da função `fat` apresentadas, por exemplo, têm como valor de retorno um número inteiro – o valor do fatorial calculado. No entanto, a possibilidade de retornar um valor nem sempre é satisfatória. Muitas vezes, precisamos transferir mais do que um valor para a função que chama, e isso não pode ser feito com o retorno explícito de valores.

Para ilustrar essa discussão, vamos inicialmente considerar uma função muito simples que calcula a soma de dois valores inteiros. Uma implementação dessa função e um exemplo de seu uso são mostrados a seguir:

```
#include <stdio.h>

int soma (int a, int b)
{
  int c;
  c = a + b;
  return c;
}
```

```
int main (void)
{
  int s;

  s = soma(3,5);
  printf("Soma = %d\n", s);

  return 0;
}
```

Esse exemplo não apresenta nenhuma dificuldade, pois o resultado da soma pode ser retornado explicitamente. O problema aparece quando desejamos que a função resulte em mais de um valor. Por exemplo, vamos considerar agora uma função para calcular a soma e o produto de dois números. Uma forma INCORRETA de implementar essa função é ilustrada a seguir:

```
/* função somaprod (versão errada) */

#include <stdio.h>

void somaprod (int a, int b, int c, int d)
{
  c = a + b;
  d = a * b;
}

int main (void)
{
  int s, p;

  somaprod(3,5,s,p);
  printf("soma = %d  produto = %d\n", s, p);

  return 0;
}
```

Como sabemos, esse código não funciona como esperado. Serão impressos valores "lixo", pois `s` e `p` não foram inicializados na função `main`, e seus valores não são alterados. Alterados são os valores de `c` e `d` dentro da função `somaprod`, mas eles não representam as variáveis da função `main` e são apenas inicializados com os valores de `s` e `p`.

Como fazemos então para que a função que chama tenha acesso aos dois valores calculados? Para resolver esse problema em C, é preciso entender antes o conceito de ponteiro para variáveis.

Variáveis do tipo ponteiro

A linguagem C permite o armazenamento e a manipulação de valores de endereços de memória. Para cada tipo existente, há um tipo ponteiro capaz de armaze-

nar endereços de memória em que existem valores do tipo correspondente armazenados. Por exemplo, quando escrevemos:

```
int a;
```

declaramos uma variável de nome a que pode armazenar valores inteiros. Automaticamente, reserva-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes).

Assim como declaramos variáveis para armazenar inteiros, podemos declarar variáveis, as quais em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória em que há valores inteiros armazenados. A linguagem C não reserva uma palavra especial para a declaração de ponteiros; usamos a mesma palavra do tipo com os nomes das variáveis precedidos pelo caractere *. Então, podemos escrever:

```
int *p;
```

Nesse caso, declaramos uma variável de nome p que pode armazenar endereços de memória em que existe um inteiro armazenado. Para atribuir e acessar endereços de memória, a linguagem oferece dois operadores unários que ainda não foram discutidos aqui. O operador unário & ("endereço de"), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. O operador unário * ("conteúdo de"), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro. Para exemplificar, vamos ilustrar esquematicamente, com base em um exemplo simples, o que ocorre na pilha de execução. Consideremos o trecho de código mostrado na Figura 4.3.

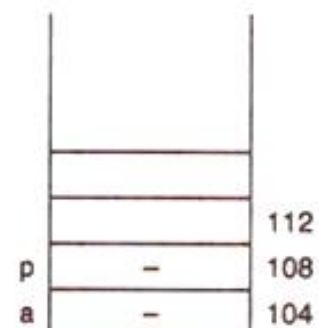



Figura 4.3 *Efeito de declarações de variáveis na pilha de execução.*

Após as declarações, as duas variáveis, a e p, armazenam valores "lixo", pois não foram inicializadas. Podemos fazer atribuições como exemplificado nos fragmentos de código da figura a seguir:

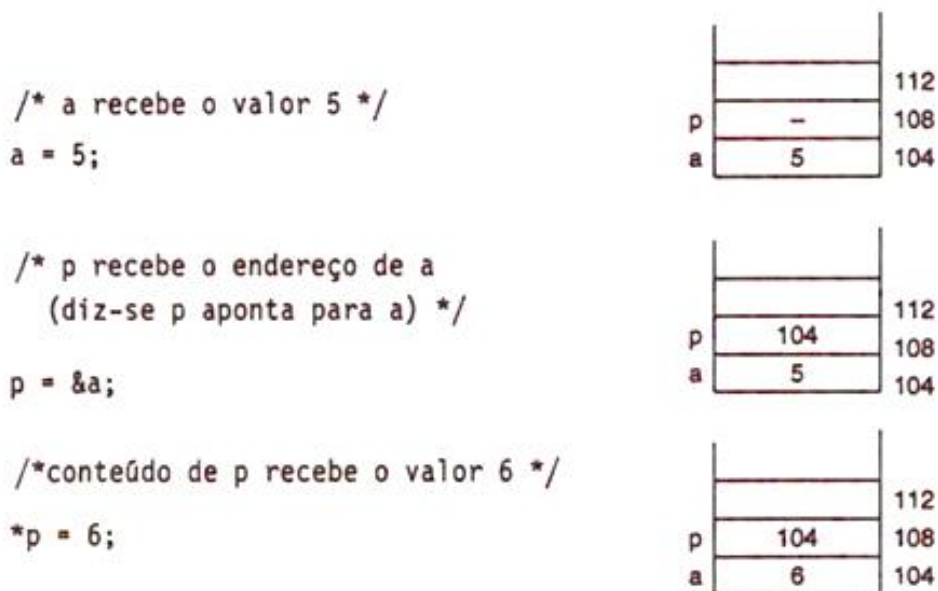



Figura 4.4 *Efeito de atribuição de variáveis na pilha de execução.*

Com as atribuições ilustradas na figura, a variável a recebe, indiretamente, o valor 6. Acessar a é equivalente a acessar *p, pois p armazena o endereço de a. Dizemos que p aponta para a, daí o nome ponteiro. Em vez de criar valores fictícios para os endereços de memória no nosso esquema ilustrativo, podemos desenhar setas graficamente, sinalizando que uma variável do tipo ponteiro aponta para uma determinada área de memória.

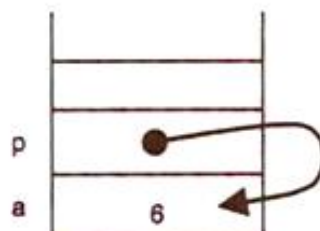



Figura 4.5 *Representação gráfica do valor de um ponteiro.*

A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C. Por outro lado, o uso indevido dessa manipulação é o maior causador de programas que "voam", isto é, não só não funcionam como, pior ainda, podem gerar efeitos colaterais não previstos.

A seguir, apresentamos outros exemplos de uso de ponteiros. O código:

```
int main ( void )
{
  int a;
  int *p;
  p = &a;
  *p = 2;
  printf(" %d ", a);
  return;
}
```

imprime o valor 2.

Agora, neste exemplo:

```
int main ( void )
{
  int a, b, *p;
  a = 2;
  *p = 3;
  b = a + (*p);
  printf(" %d ", b);
  return 0;
}
```

cometemos um ERRO típico de manipulação de ponteiros. O problema é que esse programa, embora incorreto, às vezes pode funcionar. O erro está em usar a memória apontada por `p` para armazenar o valor 3. A variável `p` não tinha sido inicializada e, portanto, tinha armazenado um valor (no caso, endereço) "lixo". Assim, a atribuição `*p = 3;` armazena 3 em um espaço de memória desconhecido, que tanto pode ser um espaço de memória não utilizado, e, nesse caso, o programa aparentemente funciona bem, quanto um espaço que armazena outras informações fundamentais – por exemplo, o espaço de memória utilizado por outras variáveis ou outros aplicativos. Nesse caso, o erro pode ter efeitos colaterais indesejados.

Portanto, só podemos preencher o conteúdo de um ponteiro se ele tiver sido devidamente inicializado, isto é, ele deve apontar para um espaço de memória para o qual já se prevê o armazenamento de valores do tipo em questão.

De maneira análoga, podemos declarar ponteiros de outros tipos:

```
float *m;
char *s;
```

Passando ponteiros para funções

Os ponteiros oferecem meios de alterar valores de variáveis ao acessá-las indiretamente. Já discutimos que as funções não podem alterar diretamente valores de variáveis da função que fez a chamada. No entanto, se passarmos para uma função os valores dos endereços de memória em que suas variáveis estão armazenadas, essa função pode alterar, indiretamente, os valores das variáveis da função que a chamou.

Para ilustrar a discussão, vamos retomar o exemplo de uma função para calcular a soma e o produto de dois números inteiros. A solução para esse problema é fazer com que a função `somaprod` receba os endereços das variáveis da função `main` e, assim, alterar seus valores indiretamente. A seguir, é ilustrada uma implementação com base nessa estratégia:

```
/* função somaprod (versão CORRETA) */

#include <stdio.h>

void somaprod (int a, int b, int *p, int *q)
{
  *p = a + b;
  *q = a * b;
}

int main (void)
{
  int s, p;

  somaprod(3,5,&s,&p);
  printf("Soma = %d  Produto = %d\n", s, p);

  return 0;
}
```

Devemos notar que a função somaprod citada não retorna explicitamente nenhum valor (é uma função do tipo void). No entanto, ela recebe o endereço de duas variáveis, armazena os valores calculados nesses endereços de memória e altera, por conseguinte, os valores das variáveis originais. A Figura 4.6 ilustra a execução do programa, mostrando o uso da memória. Assim, conseguimos o efeito desejado.

Como exemplo adicional, podemos considerar uma função que troca os valores entre duas variáveis dadas. Para que os valores das variáveis da função principal sejam alterados (trocados) dentro da função auxiliar que faz a troca, precisamos passar para a função os endereços das variáveis. O código a seguir ilustra essa implementação.

```
/* função troca */
#include <stdio.h>

void troca (int *px, int *py )
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}

int main ( void )
{
  int a = 5, b = 7;
```

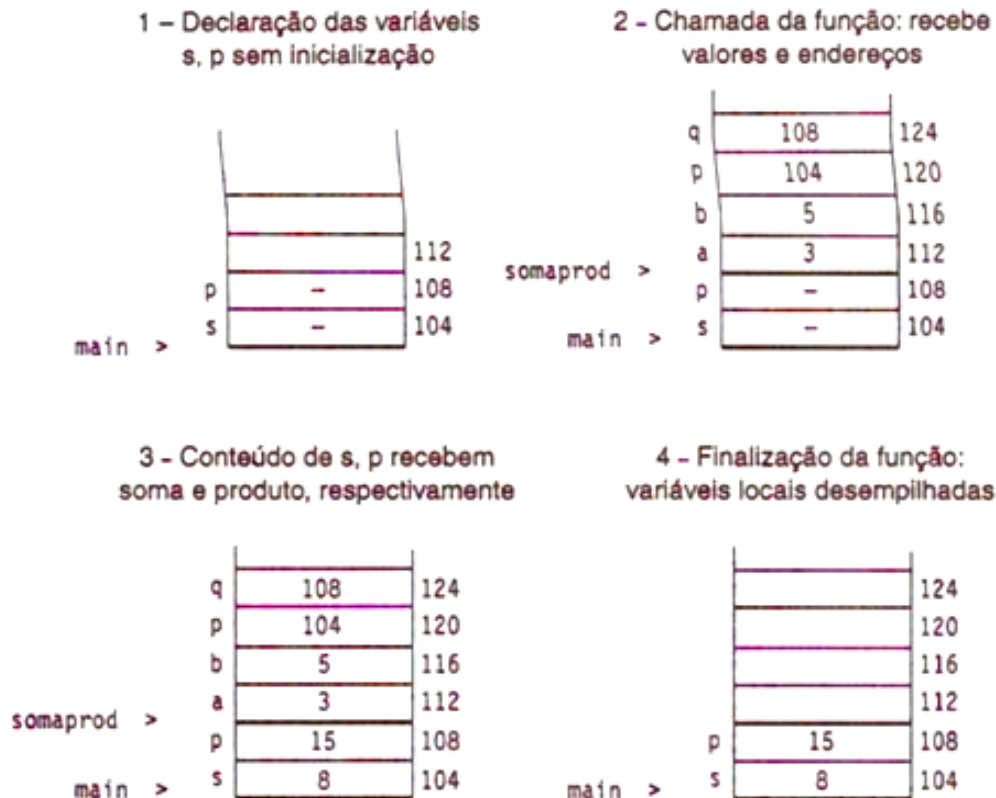



Figura 4.6 *Passo a passo da função que calcula a soma e o produto.*

```
   troca(&a, &b);      /* passamos os endereços das variáveis */
   printf("%d %d \n", a, b);
   return 0;
}
```

A Figura 4.7 ilustra a execução desse programa, mostrando o uso da memória.

Agora fica explicado por que passamos o endereço das variáveis para a função scanf, pois, caso contrário, a função não conseguiria devolver os valores lidos. De fato, sempre que desejarmos alterar um valor de uma variável da função que chama dentro da função chamada, devemos passar o endereço da variável. Assim, a função chamada tem acesso ao espaço de memória da variável e pode alterar seu valor.

Variáveis globais

Existe uma outra forma de fazer a comunicação entre funções, que consiste no uso de variáveis globais. Se uma variável é declarada fora do corpo das funções, ela é dita global. Uma variável global é visível a todas as funções subseqüentes. As variáveis globais não são armazenadas na pilha de execução, portanto não deixam de existir quando a execução de uma função termina; elas existem enquanto o programa estiver sendo executado.

1 - Declaração das variáveis : a, b

2 - Chamada da função: passa endereços

3 - Declaração da variável local: temp

4 - temp recebe conteúdo de px

5 - Conteúdo de px recebe conteúdo de py

6 - Conteúdo de py recebe temp

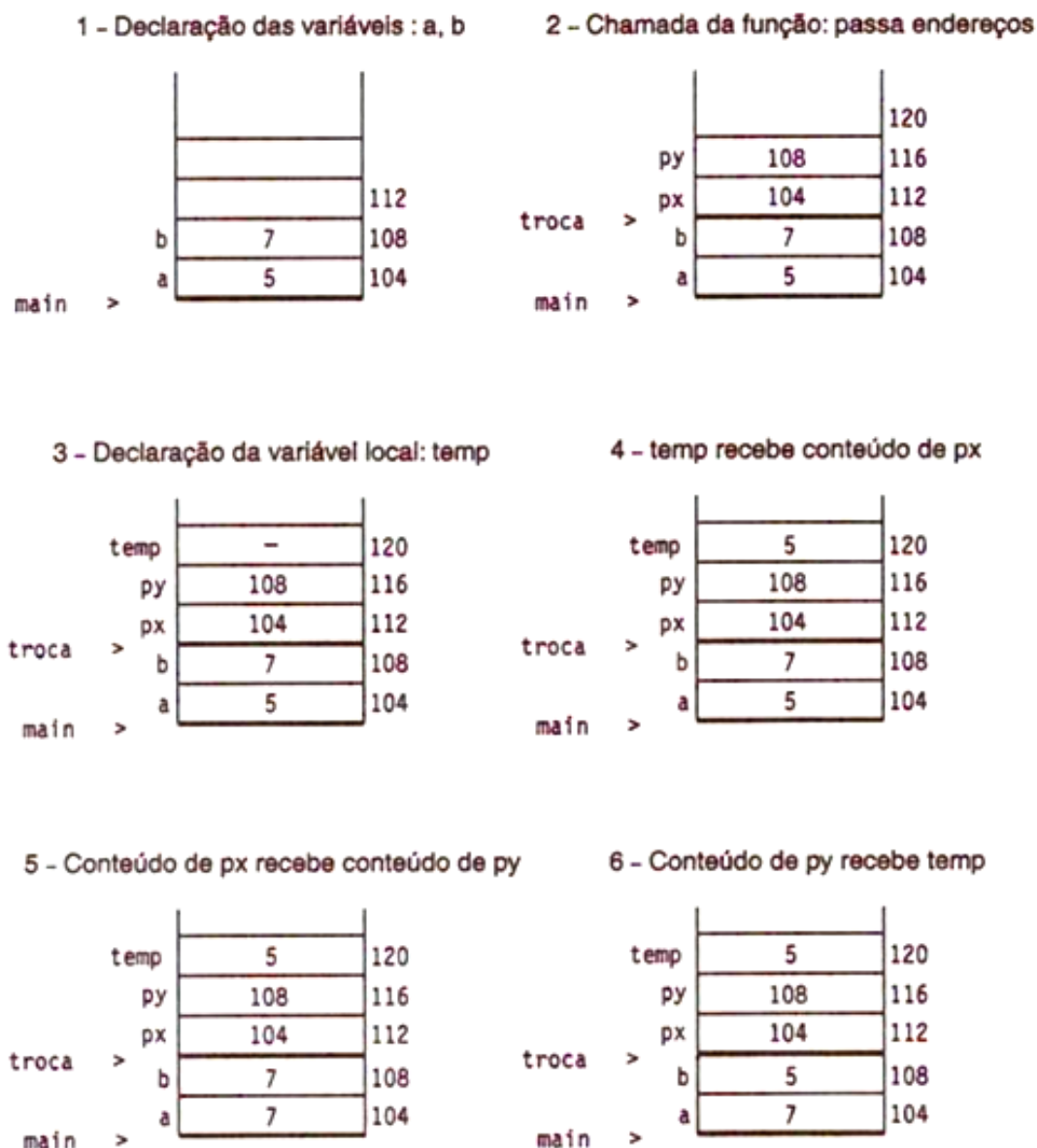



Figura 4.7 *Passo a passo da função que troca dois valores.*

Se uma determinada variável global é visível em duas funções, ambas podem acessar e/ou alterar o valor da variável diretamente. Por exemplo, podemos reescrever o código para cálculo da soma e do produto entre valores com o uso de variáveis globais, a fim de armazenar os resultados:

```
#include <stdio.h>

int s, p;  /* variáveis globais */

void somaprod (int a, int b)
{
  s = a + b;
  p = a * b;
}
```

```
int main (void)
{
  int x, y;

  scanf("%d %d", &x, &y);
  somaprod(x,y);
  printf("Soma = %d  produto = %d\n", s, p);

  return 0;
}
```

Salientamos, no entanto, que o uso de variáveis globais em um programa deve ser feito com critério, pois podemos criar um alto grau de interdependência entre as funções, o que dificulta o entendimento e a reutilização do código. Nos nossos exemplos, vamos evitar o uso de variáveis globais.

Variáveis estáticas

Podemos declarar variáveis estáticas dentro de funções. Nesse caso, as variáveis também não são armazenadas na pilha, mas sim numa área de memória estática que existe enquanto o programa está sendo executado. Ao contrário das variáveis locais (ou automáticas), que existem apenas enquanto a função à qual pertencem estiver sendo executada, as estáticas, assim como as globais, continuam existindo mesmo antes ou depois de a função ser executada. No entanto, de modo diferente das variáveis globais, uma variável estática declarada dentro de uma função só é visível dentro dessa função. Uma utilização importante de variáveis estáticas dentro de funções é quando se necessita recuperar o valor de uma variável atribuída na última vez em que a função foi executada.

Para exemplificar a utilização de variáveis estáticas declaradas dentro de funções, consideremos uma função que serve para imprimir números reais. A característica dessa função é que ela imprime um número por vez, separando-os por espaços em branco e colocando, no máximo, cinco números por linha. Com isso, do primeiro ao quinto número são impressos na primeira linha, do sexto ao décimo na segunda, e assim por diante. Uma possível implementação dessa função é mostrada a seguir:

```
void imprime ( float a )
{
  static int n = 1;

  printf(" %f   ", a);
  if ((n % 5) == 0) printf(" \n ");
  n++;
}
```

Na primeira vez em que essa função for executada, a variável n terá valor inicial 1, sendo incrementado para 2. Na segunda vez em que a função for executada, o valor de n será inicialmente 2 (preservado desde a última execução da função) e assim por diante.

Se uma variável estática não for inicializada de forma explícita na declaração, ela é automaticamente inicializada com zero. (As variáveis globais também são, por padrão, inicializadas com zero.)

Variáveis globais também podem ser declaradas como estáticas. Nesse caso, elas são visíveis para todas as funções subseqüentes, mas não podem ser acessadas por funções definidas em outros arquivos. De maneira análoga, as funções também podem ser declaradas como sendo estáticas, não podendo ser acessadas (chamadas) por funções definidas em outros arquivos. O uso de variáveis globais e funções estáticas é necessário quando estamos trabalhando com vários módulos, assunto que será abordado mais adiante.

Recursividade

As funções podem ser chamadas recursivamente, isto é, dentro do corpo de uma função podemos chamar novamente a própria função. Se uma função A chama a própria função A, dizemos que ocorre uma recursão direta. Se uma função A chama uma função B que, por sua vez, chama A, temos uma recursão indireta. Diversas implementações ficam muito mais fáceis com a recursividade. Por outro lado, implementações não recursivas tendem a ser mais eficientes.

Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução. Assim, mesmo quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

As implementações recursivas devem ser pensadas conforme a definição recursiva do problema que desejamos resolver. Por exemplo, o valor do fatorial de um número pode ser definido de forma recursiva:

$$n! = \begin{cases} 1, \text{ se } n = 0 \\ n \times (n-1)!, \text{ se } n > 0 \end{cases}$$

Considerando esta definição, fica muito simples pensar na implementação recursiva de uma função que calcula e retorna o fatorial de um número.

```
/* Função recursiva para cálculo do fatorial */

int fat (int n)
{
  if (n==0)
     return 1;
  else
     return n*fat(n-1);
}
```

Pré-processador e macros

Um código C, antes de ser compilado, passa por um pré-processador. Esse pré-processador reconhece determinadas diretivas e altera o código para, então, enviá-lo ao compilador.

Uma das diretivas reconhecidas pelo pré-processador, e já utilizada nos nossos exemplos, é `#include`. Ela é seguida por um nome de arquivo, e o pré-processador a substitui pelo corpo do arquivo especificado. É como se o texto do arquivo incluído fizesse parte do código-fonte.

É importante observar que quando o nome do arquivo a ser incluído é envolto por aspas ("*arquivo*"), o pré-processador tipicamente procura o arquivo primeiro no diretório local (em geral denominado diretório de trabalho) e, caso não o encontre, o procura nos diretórios de *include*, especificados para compilação. Se o arquivo é colocado entre os sinais de menor e maior (<*arquivo*>), o pré-processador não procura o arquivo no diretório local (os arquivos da biblioteca padrão de C devem ser incluídos com < >).

Outra diretiva de pré-processamento, muito utilizada e que será agora discutida, é a diretiva de definição. Por exemplo, uma função para calcular a área de um círculo pode ser escrita da seguinte forma:

```
#define PI  3.14159F

float area (float r)
{
  float a = PI * r * r;
  return a;
}
```

Nesse caso, antes da compilação, toda ocorrência da palavra `PI` (desde que não esteja envolvida por aspas) será trocada pelo número `3.14159F`. O uso de diretivas de definição para representar constantes simbólicas é fortemente recomendável, pois facilita a manutenção e acrescenta clareza ao código.

A linguagem C permite ainda a utilização da diretiva de definição com parâmetros. É válido escrever, por exemplo:

```
#define MAX(a,b)    ((a) > (b) ? (a) : (b))
```

assim, se após a definição existir uma linha de código com o trecho:

```
v = 4.5;
c = MAX(v, 3.0);
```

o compilador verá:

```
v = 4.5;
c = ((v) > (3.0) ? (v) : (3.0));
```

Essas definições com parâmetros recebem o nome de macros. Devemos ter muito cuidado na definição de macros. Mesmo um erro de sintaxe pode ser difícil de ser detectado, pois o compilador indicará um erro na linha em que se utiliza a macro e não na linha de definição da macro (na qual efetivamente se encontra o erro). Outros efeitos colaterais de macros mal definidas podem ser ainda piores. Por exemplo, no código a seguir:

```
#include <stdio.h>

#define DIF(a,b)    a - b

int main (void)
{
  printf(" %d ", 4 * DIF(5,3));
  return 0;
}
```

o resultado impresso é 17 e não 8, como poderia ser esperado. A razão é simples, pois para o compilador (fazendo a substituição da macro) está escrito:

```
  printf(" %d ", 4 * 5 - 3);
```

e a multiplicação tem precedência sobre a subtração. Nesse caso, parênteses em volta da macro resolveriam o problema. No entanto, neste outro exemplo que envolve a macro com parênteses:

```
#include <stdio.h>

#define PROD(a,b)    (a * b)

int main (void)
{
  printf(" %d ", PROD(3+4, 2));
  return 0;
}
```

o resultado é 11 e não 14. A macro corretamente definida seria:

```
#define PROD(a,b)   ((a) * (b))
```

Concluímos, portanto, que, como regra básica para a definição de macros, devemos envolver cada parâmetro, além da macro como um todo, com parênteses.

5

Vetores e alocação dinâmica

Neste capítulo, discutiremos a forma mais primitiva de armazenar um conjunto de dados na memória do computador. Para motivar a discussão, vamos considerar inicialmente um exemplo simples de um programa para calcular a média aritmética de *n* valores reais fornecidos pelo usuário via teclado. Esse programa pode primeiro capturar o número de valores a serem fornecidos e, então, capturar os valores para efetuar o cálculo da média. Como sabemos, a média aritmética de um conjunto de valores é dada por:

$$m = \frac{\sum x}{N}$$

Mostramos a seguir um programa para efetuar esse cálculo:

```
/* Cálculo da média de n números reais */

#include <stdio.h>

int main ( void )
{
  int i;
  int n;              /* número de valores a serem capturados */
  float med = 0.0f;  /* valor da média */

  /* leitura do número de valores */
  scanf("%d", &n);
```

```
  /* leitura do conjunto de valores e cálculo do somatório */
  for (i = 0; i < n; i++) {
     float v;              /* variável para armazenar valor lido */
     scanf("%f", &v);      /* lê cada valor */
     med = med + v;        /* acumula soma dos valores */
  }

  /* cálculo da média */
  med = med / n;

  /* exibição do resultado */
  printf("Valor da media = %f\n", med);

  return 0;
}
```

Como vemos, esse exemplo é simples e pode ser construído sem precisar armazenar o conjunto de valores, pois o cálculo da soma dos valores pode ser feito enquanto os valores são capturados. Em muitas aplicações, entretanto, necessitamos armazenar o conjunto de valores na memória do computador para depois efetuar computações com esses valores. Como exemplo, vamos considerar que, além da média, desejássemos também calcular a variância do conjunto de valores fornecidos no exemplo anterior. Os valores da média e da variância são dados pelas fórmulas:

$$m = \frac{\sum x}{N}, v = \frac{\sum (x - m)^2}{N}$$

Portanto, precisamos ter o valor da média para então calcular o valor da variância. Dessa forma, temos de armazenar o conjunto de valores capturados na memória, pois não é possível calcular a variância durante a leitura dos dados. Para tanto, introduziremos o conceito de vetores.

Vetores

A forma mais simples de estruturar um conjunto de dados é por meio de vetores. Como a maioria das linguagens de programação, C permite a definição de vetores. Definimos um vetor em C da seguinte forma:

```
int v[10];
```

Essa declaração diz que v é um vetor de inteiros dimensionado com 10 elementos, isto é, reservamos um espaço de memória contínuo para armazenar 10 valores inteiros. Assim, se cada `int` ocupa 4 bytes, a declaração reserva um espaço de memória de 40 bytes, como ilustra a Figura 5.1.

O acesso a cada elemento do vetor é feito por meio de uma indexação da variável v. Observamos que, em C, a indexação de um vetor varia de *0* a *n-1*, onde *n*

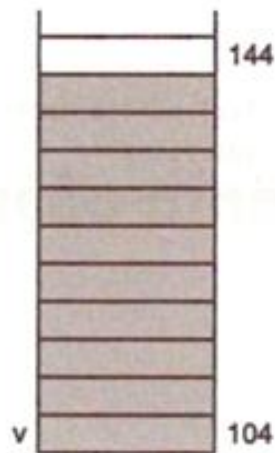



Figura 5.1 *Espaço de memória de um vetor de 10 elementos inteiros.*

representa a dimensão do vetor. Assim, para a declaração do vetor ilustrada aqui, temos:

`v[0]` → *acessa o primeiro elemento de v*
`v[1]` → *acessa o segundo elemento de v*
...
`v[9]` → *acessa o último elemento de v*

Mas:

`v[10]` → *está ERRADO (invasão de memória)*

Para exemplificar o uso de vetores, vamos voltar ao problema do cálculo da média e da variância de um conjunto de valores. Ainda para simplificar, vamos considerar que desejamos calcular esses valores para um conjunto de 10 números reais.

Uma possível implementação desse programa com a utilização de vetores é apresentada a seguir. Os valores são lidos e armazenados no vetor. Depois, efetuamos os cálculos da média e da variância sobre o conjunto de valores armazenado.

```
/* Cálculo da média e da variância de 10 números reais */

#include <stdio.h>

int main ( void )
{
  float v[10];       /* declara vetor com 10 elementos */
  float med, var;    /* variáveis para a média e a variância */
  int i;             /* variável usada como índice do vetor */
```

```
  /* leitura dos valores */
  for (i = 0; i < 10; i++)          /* faz índice variar de 0 a 9 */
     scanf("%f", &v[i]);            /* lê cada elemento do vetor */

  /* cálculo da média */
  med = 0.0f;                          /* inicializa média com zero */
  for (i = 0; i < 10; i++)
     med = med + v[i];                /* acumula soma dos elementos */
  med = med / 10;                     /* calcula a média */

  /* cálculo da variância */
  var = 0.0f;                           /* inicializa com zero */
  for ( i = 0; i < 10; i++ )
    var = var+(v[i]-med)*(v[i]-med); /* acumula */
  var = var / 10;                      /* calcula a variância */

  /* exibição do resultado */
  printf ( "Media = %f   Variancia = %f  \n", med, var );
  return 0;
}
```

Devemos observar que passamos para a função `scanf` o endereço de cada elemento do vetor (`&v[i]`), pois desejamos o armazenamento dos valores capturados nos elementos do vetor. Se `v[i]` representa o *(i+1)-ésimo* elemento do vetor, `&v[i]` representa o endereço de memória em que esse elemento está armazenado.

Na verdade, existe uma associação forte entre vetores e ponteiros, pois se existe a declaração:

```
int v[10];
```

o símbolo `v`, o qual representa o vetor, é uma constante que representa seu endereço inicial, isto é, `v`, sem indexação, aponta para o primeiro elemento do vetor.

A linguagem C também suporta aritmética de ponteiros. Podemos somar e subtrair ponteiros, desde que o valor do ponteiro resultante aponte para dentro da área reservada para o vetor. Se `p` representa um ponteiro para um inteiro, `p+1` representa um ponteiro para o próximo inteiro armazenado na memória, isto é, o valor de `p` é incrementado em 4 (mais uma vez assumindo que um inteiro tem 4 bytes). Com isso, em um vetor temos as seguintes equivalências:

`v+0` → *aponta para o primeiro elemento do vetor*
`v+1` → *aponta para o segundo elemento do vetor*
`v+2` → *aponta para o terceiro elemento do vetor*
...
`v+9` → *aponta para o décimo elemento do vetor*

Portanto, escrever &v[i] é equivalente a escrever (v+i). De maneira análoga, escrever v[i] é equivalente a escrever *(v+i) (é lógico que a forma indexada é mais clara e adequada). Devemos notar que o uso da aritmética de ponteiros aqui é perfeitamente válido, pois os elementos dos vetores são armazenados de forma contínua na memória.

Os vetores também podem ser inicializados na declaração:

```
int v[5] = { 5, 10, 15, 20, 25 };
```

ou simplesmente:

```
int v[ ] = { 5, 10, 15, 20, 25 };
```

Nesse último caso, a linguagem dimensiona o vetor pelo número de elementos inicializados.

Passagem de vetores para funções

Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor. Se passamos um valor de endereço, a função chamada deve ter um parâmetro do tipo ponteiro para armazenar esse valor. Assim, se passarmos para uma função um vetor de int, devemos ter um parâmetro do tipo int*, capaz de armazenar endereços de inteiros. Salientamos que a expressão "passar um vetor para uma função" deve ser interpretada como "passar o endereço inicial do vetor". Os elementos do vetor não são copiados para a função, o argumento copiado é apenas o endereço do primeiro elemento.

Para exemplificar, vamos modificar o código do exemplo anterior, usando funções separadas para o cálculo da média e da variância. (Aqui, usamos ainda os operadores de atribuição += para acumular as somas.)

```
/* Cálculo da média e da variância de 10 reais (segunda versão) */

#include <stdio.h>

/* Função para cálculo da média */
float media (int n, float* v)
{
  int i;
  float s = 0.0f;
  for (i = 0; i < n; i++)
     s += v[i];
  return s/n;
}
```

```
/* Função para cálculo da variância */
float variancia (int n, float* v, float m)
{
  int i;
  float s = 0.0f;
  for (i = 0; i < n; i++)
     s += (v[i] - m) * (v[i] - m);
  return s/n;
}

int main ( void )
{
  float v[10];
  float med, var;
  int i;
  /* leitura dos valores */
  for ( i = 0; i < 10; i++ )
     scanf("%f", &v[i]);

  med = media(10,v);
  var = variancia(10,v,med);
  printf ( "Media = %f   Variancia = %f  \n", med, var);
  return 0;
}
```

Observamos ainda que, como é passado para a função o endereço do primeiro elemento do vetor (e não os elementos propriamente ditos), podemos alterar os valores dos elementos do vetor dentro da função. O exemplo a seguir ilustra este fato: dentro de uma função incrementamos todos os elementos em uma unidade.

```
/* Incrementa elementos de um vetor */

#include <stdio.h>

void incr_vetor ( int n, int *v )
{
  int i;
  for (i = 0; i < n; i++)
     v[i]++;
}
int main ( void )
{
  int a[ ] = {1, 3, 5};
  incr_vetor(3, a);
  printf("%d %d %d \n", a[0], a[1], a[2]);
  return 0;
}
```

A saída do programa é 2 4 6, pois os elementos do vetor serão incrementados dentro da função, alterando os valores originais do vetor.

Alocação dinâmica

Até aqui, na declaração de um vetor, foi preciso dimensioná-lo, o que nos obrigava a saber, de antemão, quanto espaço seria necessário, isto é, tínhamos de prever o número máximo de elementos no vetor durante a codificação. Esse pré-dimensionamento é um fator limitante. Por exemplo, se desenvolvermos um programa para calcular a média e a variância das notas de uma prova, teremos de prever o número máximo de alunos. Uma solução é dimensionar o vetor com um número absurdamente alto, para não termos limitações no momento da utilização do programa. No entanto, isso levaria a um desperdício de memória, o que é inaceitável em diversas aplicações. Se, por outro lado, formos modestos no pré-dimensionamento do vetor, o uso do programa fica muito limitado, pois não conseguiríamos tratar turmas com um número de alunos maior do que o previsto.

Felizmente, a linguagem C oferece meios de requisitar espaços de memória em tempo de execução. Dizemos que podemos alocar memória dinamicamente. Com esse recurso, nosso programa para o cálculo da média e variância discutido antes pode, em tempo de execução, consultar o número de alunos da turma e então fazer a alocação do vetor dinamicamente, sem desperdício de memória.

Uso da memória

Informalmente, podemos dizer que existem três maneiras de reservar espaço de memória para o armazenamento de informações:

A primeira é usar variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado.

A segunda maneira é usar variáveis locais. Nesse caso, como já discutimos, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Por esse motivo, a função que chama não pode fazer referência ao espaço local da função chamada. As variáveis globais ou locais podem ser simples ou vetores. Para os vetores, precisamos informar o número máximo de elementos; caso contrário, o compilador não saberia o tamanho do espaço a ser reservado.

A terceira maneira de reservar memória é requisitar ao sistema, em tempo de execução, um espaço de um determinado tamanho. Esse espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado pelo programa. Por isso, podemos alocar dinamicamente um espaço de memória em uma função e acessá-lo em outra. A partir do momento em que liberarmos o espaço,

ele estará disponibilizado para outros usos e não podemos mais acessá-lo. Se o programa não liberar um espaço alocado, ele será automaticamente liberado quando a execução do programa terminar.

Na Figura 5.2, apresentamos um esquema didático que ilustra de maneira fictícia a distribuição do uso da memória pelo sistema operacional.[1] Quando requisitamos ao sistema operacional para executar um determinado programa, o código em linguagem de máquina do programa deve ser carregado na memória, conforme discutido no primeiro capítulo. O sistema operacional reserva também os espaços necessários para armazenar as variáveis globais (e estáticas) existentes no programa. O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Cada vez que uma determinada função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Esse espaço pertence à pilha de execução e, quando a função termina, é desempilhado. A parte da memória não ocupada pela pilha de execução pode ser requisitada dinamicamente. Se a pilha tentar crescer além do espaço disponível existente, dizemos que ela "estourou", e o programa é abortado com erro. Da mesma forma, se o espaço de memória livre for menor do que o espaço requisitado dinamicamente, a alocação não é feita, e o programa pode prever um tratamento de erro adequado (por exemplo, podemos imprimir a mensagem "Memória insuficiente" e interromper a execução do programa).

Figura 5.2 *Alocação esquemática de memória.*

Funções da biblioteca padrão

Existem funções, presentes na biblioteca padrão *stdlib*, que permitem alocar e liberar memória dinamicamente. A função básica para alocar memória é `malloc`. Ela recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Para exemplificar, vamos considerar a alocação dinâmica de um vetor de inteiros com 10 elementos. Como a função `malloc` tem como valor de retorno o endereço da área alocada e, nesse exemplo, desejamos armazenar valores inteiros nessa área, devemos declarar um ponteiro de inteiro para receber o endereço inicial do espaço alocado. O trecho de código então seria:

```
int *v;

v = malloc(10*4);
```

Após esse comando, se a alocação for bem-sucedida, `v` armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros. Podemos, então, tratar `v` como tratamos um vetor declarado estaticamente, pois, se `v` aponta para o início da área alocada, sabemos que `v[0]` acessa o espaço para o primeiro elemento a ser armazenado, `v[1]` acessa o segundo, e assim por diante (até `v[9]`).

No exemplo mostrado, consideramos que um inteiro ocupa 4 bytes. Para ficarmos independentes de compiladores e máquinas, usamos o operador `sizeof( )`.

```
v = malloc(10*sizeof(int));
```

Além disso, devemos salientar que a função `malloc` é usada para alocar espaço para armazenar valores de qualquer tipo. Por esse motivo, `malloc` retorna um ponteiro genérico, para um tipo qualquer, representado por `void*`, que pode ser convertido automaticamente pela linguagem para o tipo apropriado na atribuição. No entanto, é comum fazer a conversão explicitamente, utilizando o operador de molde de tipo (*cast*)[2]. O comando para a alocação do vetor de inteiros fica então:

```
v = (int *) malloc(10*sizeof(int));
```

A Figura 5.3 ilustra de maneira esquemática o que ocorre na memória:

[2] A linguagem C++, por exemplo, exige a conversão explícita para o tipo correto.

Figura 5.3 *Alocação dinâmica de memória.*

Se, porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo `NULL`, definido em *stdlib.h*). Podemos cercar o erro na alocação da memória verificando o valor de retorno da função `malloc`. Por exemplo, podemos imprimir uma mensagem e abortar o programa com a função `exit`, também definida na *stdlib*.

```
…
v = (int*) malloc(10*sizeof(int));
if (v==NULL)
{
  printf("Memoria insuficiente.\n");
  exit(1);  /* aborta o programa e retorna 1 para o sist. operacional */
}
…
```

Para liberar um espaço de memória alocado dinamicamente, usamos a função `free`. Essa função recebe como parâmetro o ponteiro da memória a ser liberada. Assim, para liberar o vetor `v`, fazemos:

```
free (v);
```

Só podemos passar para a função `free` um endereço de memória que tenha sido alocado dinamicamente. Devemos lembrar ainda que não podemos acessar o espaço da memória depois de liberado.

Para exemplificar o uso da alocação dinâmica, alteraremos o programa para o cálculo da média e da variância mostrado anteriormente. Agora, o programa lê o número de valores que serão fornecidos, aloca um vetor dinamicamente, captura os valores e faz os cálculos. Somente a função principal precisa ser alterada,

pois as funções para calcular a média e a variância anteriormente apresentadas independem do fato de o vetor ter sido alocado estática ou dinamicamente.

```
/* Cálculo da média e da variância de n reais */

#include <stdio.h>
#include <stdlib.h>

...

int main ( void )
{
  int i, n;
  float *v;
  float med, var;

  /* leitura do número de valores */
  scanf("%d", &n);
  /* alocação dinâmica */
  v = (float*) malloc(n*sizeof(float));
  if (v==NULL) {
     printf("Memoria insuficiente.\n");
     return 1;
  }
  /* leitura dos valores */
  for (i = 0; i < n; i++)
     scanf("%f", &v[i]);
  med = media(n,v);
  var = variancia(n,v,med);
  printf("Media = %f    Variancia = %f  \n", med, var);
  /* libera memória */
  free(v);
  return 0;
}
```

Vetores locais a funções

Em geral, reservamos o uso de alocação dinâmica para os casos em que a dimensão do vetor é desconhecida. Quando sabemos sua dimensão, é preferível o uso de vetores declarados localmente. No entanto, devemos mais uma vez salientar que a área de memória de uma variável local só existe enquanto a função que a declara estiver sendo executada. Esse fato requer cuidado quando da utilização de vetores locais dentro de funções.

Para exemplificar, vamos considerar uma aplicação que manipula vetores algébricos no espaço tridimensional. Um vetor algébrico em 3D é representado pelas três componentes *x*, *y*, e *z*. Podemos então representar um vetor algébrico por um vetor (de C) de dimensão 3.

Vamos agora considerar a implementação de uma função que calcula o produto vetorial de dois vetores. O produto vetorial é dado por:

$$\mathbf{u} \times \mathbf{v} = \{u_y\, v_z - v_y\, u_z,\ u_z\, v_x - v_z\, u_x,\ u_x\, v_y - v_x\, u_y\}$$

Podemos pensar em uma função que recebe dois vetores como parâmetros e retorna o resultado do produto vetorial. Uma forma INCORRETA de implementar essa função é:

```
float* prod_vetorial (float* u, float* v)
{
  float p[3];
  p[0] = u[1]*v[2] - v[1]*u[2];
  p[1] = u[2]*v[0] - v[2]*u[0];
  p[2] = u[0]*v[1] - v[0]*u[1];
  return p;  /* ERRO: não podemos retornar endereço de área local */
}
```

O erro nessa implementação consiste no fato de retornarmos o valor de um endereço de memória que não estará mais disponível quando a função terminar. A variável `p` é declarada localmente, portanto essa área de memória deixa de ser válida quando a função termina. Assim, a função que chama não pode acessar a área apontada pelo valor retornado.

Uma possível solução para o problema consiste em usar alocação dinâmica. A implementação da função seria então dada por:

```
float* prod_vetorial (float* u, float* v)
{
  float *p = (float*) malloc(3*sizeof(float));
  p[0] = u[1]*v[2] - v[1]*u[2];
  p[1] = u[2]*v[0] - v[2]*u[0];
  p[2] = u[0]*v[1] - v[0]*u[1];
  return p;
}
```

Nesse caso, a implementação é válida, pois a área apontada por `p`, alocada dinamicamente, permanece válida mesmo após o término da função. Assim, a função que chama poderia acessar o ponteiro retornado. O único problema nessa solução é fazermos uma alocação dinâmica para cada chamada da função, o que, em geral, é ineficiente do ponto de vista computacional e requer que a função que chama seja responsável pela liberação do espaço alocado.

Uma outra solução consiste em requisitar que o espaço de memória para o armazenamento do resultado já seja passado pela função que chama. Assim, a fun-

ção para o cálculo do produto recebe três vetores, dois com dados de entrada e um para armazenar o resultado. Uma implementação dessa estratégia é ilustrada a seguir.

```
void prod_vetorial (float* u, float* v, float* p)
{
  p[0] = u[1]*v[2] - v[1]*u[2];
  p[1] = u[2]*v[0] - v[2]*u[0];
  p[2] = u[0]*v[1] - v[0]*u[1];
}
```

Para esse problema, esta última solução é, em geral, mais adequada, pois não envolve alocação dinâmica. Quando discutirmos tipos estruturados, vamos verificar a existência de mais uma alternativa, que nos permite retornar explicitamente as três componentes do vetor.

6

Matrizes

Discutimos no capítulo anterior a construção de conjuntos unidimensionais usando vetores. A linguagem C também permite a construção de conjuntos bi ou multidimensionais. Neste capítulo, discutiremos em detalhes a manipulação de matrizes, representadas por conjuntos bidimensionais de valores numéricos. As construções apresentadas aqui podem ser estendidas para conjuntos de dimensões maiores.

Alocação estática *versus* dinâmica

Antes de tratar das construções de matrizes, vamos recapitular alguns conceitos apresentados com vetores. A forma mais simples de declarar um vetor de inteiros em C é mostrada a seguir:

```
int v[10];
```

ou, se quisermos criar uma constante simbólica para a dimensão:

```
#define N 10
int v[N];
```

Podemos dizer que, nesses casos, os vetores são declarados "estaticamente".[1] A variável que representa o vetor é uma constante que armazena o endereço ocupado pelo primeiro elemento do vetor. Esses vetores podem ser declarados como variáveis globais ou dentro do corpo de uma função. Se declarado dentro do corpo de uma função, o vetor existirá apenas enquanto a função estiver sendo exe-

[1] O termo "estático" aqui refere-se ao fato de não usarmos alocação dinâmica.

cutada, pois o espaço de memória para o vetor é reservado na pilha de execução. Portanto, não podemos fazer referência ao espaço de memória de um vetor local de uma função que já retornou.

Uma limitação do uso de um vetor declarado estaticamente, seja como variável global ou local, é que precisamos saber de antemão a dimensão máxima do vetor. Usando alocação dinâmica, podemos determinar a dimensão do vetor em tempo de execução:

```
int* v;
...
v = (int*) malloc(n * sizeof(int));
```

Nesse fragmento de código, n representa uma variável com a dimensão do vetor, determinada em tempo de execução (podemos, por exemplo, capturar o valor de n fornecido pelo usuário). Após a alocação dinâmica, acessamos os elementos do vetor da mesma forma que os elementos de vetores criados estaticamente. Outra diferença importante: com alocação dinâmica, declaramos uma variável do tipo ponteiro que posteriormente recebe o valor do endereço do primeiro elemento do vetor, alocado dinamicamente. Nesse caso, a área de memória ocupada pelo vetor permanece válida até que seja explicitamente liberada (usando a função free). Portanto, mesmo que um vetor seja criado dinamicamente dentro da função, podemos acessá-lo depois da função ser finalizada, pois a área de memória ocupada por ele permanece válida, isto é, o vetor não está alocado na pilha de execução.

A linguagem C oferece ainda um mecanismo para realocarmos um vetor dinamicamente. Em tempo de execução, podemos verificar que a dimensão escolhida para um vetor tornou-se insuficiente (ou excessivamente grande) e necessitava de um redimensionamento. A função realloc da biblioteca padrão nos permite realocar um vetor preservando o conteúdo dos elementos, que permanecem válidos após a realocação (no fragmento de código a seguir, m representa a nova dimensão do vetor).

```
v = (int*) realloc(v, m*sizeof(int));
```

Vale salientar que, sempre que possível, optamos por trabalhar com vetores criados estaticamente. Eles tendem a ser mais eficientes, já que os vetores alocados dinamicamente têm uma indireção a mais (primeiro acessa-se o valor do endereço armazenado na variável ponteiro para então acessar o elemento do vetor).

Vetores bidimensionais – matrizes

A linguagem C permite a criação de vetores bidimensionais, declarados estaticamente. Por exemplo, para declarar uma matriz de valores reais com 4 linhas e 3 colunas, fazemos:

```
float mat[4][3];
```

Essa declaração reserva um espaço de memória necessário para armazenar os 12 elementos da matriz, que são armazenados de maneira contínua, organizados linha a linha (Figura 6.1).

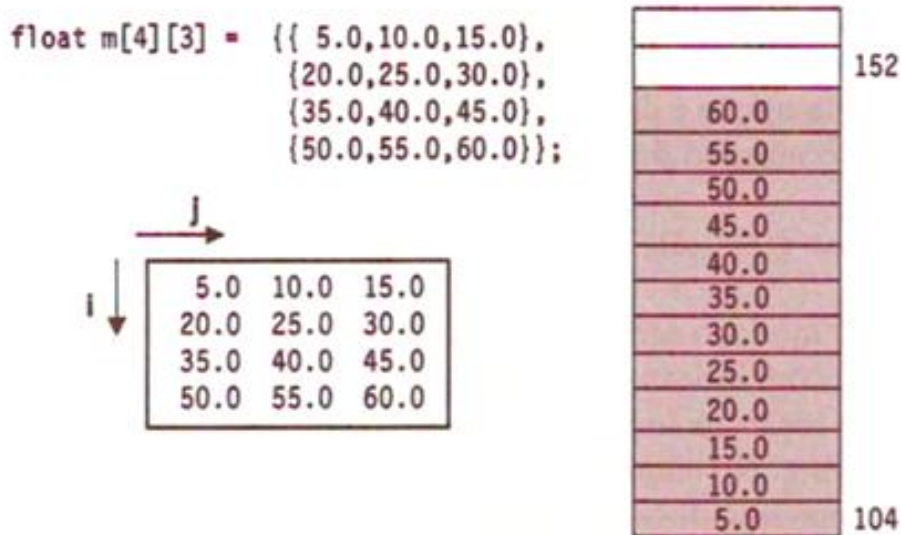



Figura 6.1 *Alocação dos elementos de uma matriz.*

Os elementos da matriz são acessados com indexação dupla: `mat[i][j]`. O primeiro índice, `i`, acessa a linha, e o segundo, `j`, acessa a coluna. Como em C a indexação começa em zero, o elemento da primeira linha e da primeira coluna é acessado por `mat[0][0]`. Após a declaração estática de uma matriz, a variável que representa a matriz, `mat` no exemplo acima, representa um ponteiro para o primeiro "vetor-linha", composto por 3 elementos. Com isto, `mat[1]` aponta para o primeiro elemento do segundo "vetor-linha", e assim por diante.

As matrizes também podem ser inicializadas na declaração:

```
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
```

Ou podemos inicializar seqüencialmente:

```
float mat[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

O número de elementos por linha pode ser omitido numa inicialização, mas o número de colunas deve ser sempre fornecido:

```
float mat[ ][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Passagem de matrizes para funções

Conforme já mencionado, uma matriz criada estaticamente é representada por um ponteiro para um "vetor-linha" com o número de elementos da linha. Quan-

do passamos uma matriz para uma função, o parâmetro da função deve ser desse tipo. Infelizmente, a sintaxe para representá-lo é obscura. O protótipo de uma função que recebe a matriz declarada anteriormente seria:

```
void f (..., float (*mat)[3], ...);
```

Uma segunda opção é declarar o parâmetro como matriz, com a possibilidade de omitir o número de linhas:[2]

```
void f (..., float mat[ ][3], ...);
```

De qualquer modo, o acesso aos elementos da matriz dentro da função é feito da forma usual, com indexação dupla.

Na próxima seção, examinaremos maneiras de trabalhar com matrizes alocadas dinamicamente. No entanto, vale salientar que recomendamos, quando possível, o uso de matrizes alocadas estaticamente. Em diversas aplicações, as matrizes têm dimensões fixas e não justificam a criação de estratégias para trabalhar com alocação dinâmica. Transformações algébricas no espaço 3D, por exemplo, são comumente representadas por matrizes 4 por 4. Nesses casos, é muito mais simples definir as matrizes estaticamente (`float mat[4][4];`), uma vez que sabemos de antemão as dimensões a serem usadas.

Matrizes dinâmicas

As matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores: precisamos saber de antemão suas dimensões. Se as dimensões só são conhecidas em tempo de execução, devemos utilizar alocação dinâmica. O problema encontrado é que a linguagem C só permite alocar dinamicamente conjuntos unidimensionais. Para trabalhar com matrizes alocadas dinamicamente, temos de criar abstrações conceituais com vetores para representar conjuntos bidimensionais. Nesta seção, discutiremos duas estratégias distintas para representar matrizes alocadas dinamicamente.

Matriz representada por um vetor simples

Concretamente, para representar uma matriz, precisamos de um espaço de memória suficiente para armazenar seus elementos. Podemos, então, adotar a estratégia de armazenar os elementos da matriz em um vetor simples. Assim, reservamos as primeiras posições do vetor para armazenar os elementos da primeira linha, seguidos dos elementos da segunda linha, e assim por diante. Conceitualmente, traba-

[2] Isso também vale para vetores. Um protótipo de uma função que recebe um vetor como parâmetro pode ser dado por: `void f (..., float v[ ], ...);`.

lharemos com um conjunto bidimensional, mas, de fato, temos um vetor unidimensional. Portanto, temos de criar uma disciplina para acessar os elementos da matriz, representada conceitualmente. A estratégia de endereçamento para acessar os elementos é a seguinte: se quisermos acessar o que seria o elemento `mat[i][j]` de uma matriz, devemos acessar o elemento `v[k]`, com `k=i*n+j`, onde `n` representa o número de colunas da matriz, conforme ilustrado na Figura 6.2.

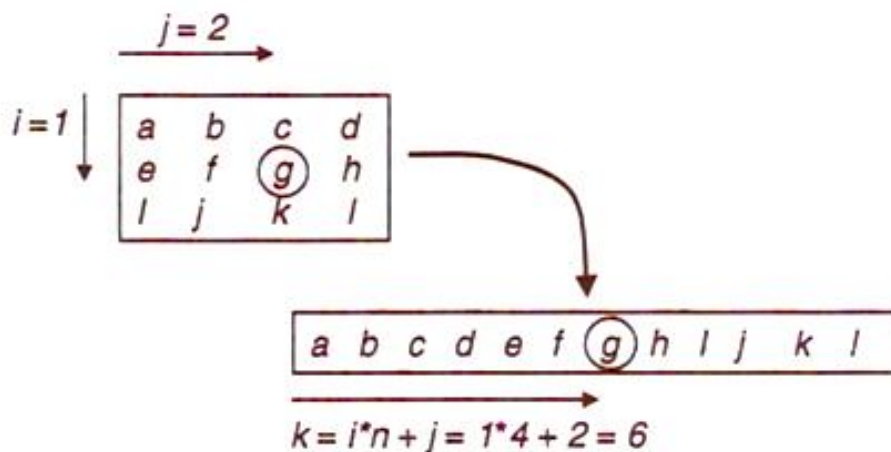



Figura 6.2 *Matriz representada por vetor simples.*

Essa conta de endereçamento é intuitiva: se quisermos acessar elementos da terceira (`i=2`) linha da matriz, temos de pular duas linhas de elementos (`i*n`) e depois indexar o elemento da linha com `j`.

Com essa estratégia, a alocação da "matriz" recai em uma alocação de vetor com `m*n` elementos, onde `m` e `n` representam as dimensões da matriz.

```
float *mat;        /* matriz representada por um vetor */
...
mat = (float*) malloc(m*n*sizeof(float));
...
```

No entanto, somos obrigados a usar uma notação desconfortável, `v[i*n+j]`, para acessar os elementos, o que pode deixar o código pouco legível.

Matriz representada por um vetor de ponteiros

Vamos agora apresentar outra estratégia para trabalhar com matrizes dinâmicas que usam vetores simples. Nesta segunda estratégia, cada linha da matriz é representada por um vetor independente. A matriz é então representada por um vetor de vetores, ou vetor de ponteiros, no qual cada elemento armazena o endereço do primeiro elemento de cada linha. A Figura 6.3 ilustra o arranjo da memória utilizado nessa estratégia.

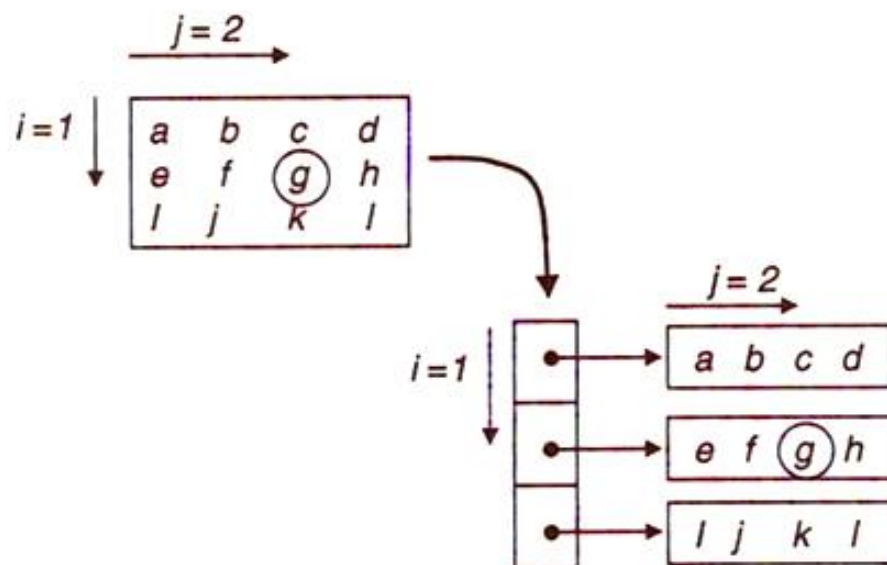



Figura 6.3 *Matriz com vetor de ponteiros.*

A alocação da matriz agora é mais elaborada. Primeiro, temos de alocar o vetor de ponteiros. Em seguida, alocamos cada uma das linhas da matriz, atribuindo seus endereços aos elementos do vetor de ponteiros criado. O fragmento de código a seguir ilustra essa codificação:

```
int i;
float **mat;      /* matriz representada por um vetor de ponteiros */
...
mat = (float**) malloc(m*sizeof(float*));
for (i=0; i<m; i++)
  mat[i] = (float*) malloc(n*sizeof(float));
```

A grande vantagem dessa estratégia é o acesso aos elementos ser feito da mesma forma que quando temos uma matriz criada estaticamente, pois, se `mat` representa uma matriz alocada segundo essa estratégia, `mat[i]` representa o ponteiro para o primeiro elemento da linha `i`, e, conseqüentemente, `mat[i][j]` acessa o elemento da coluna `j` da linha `i`.

A liberação do espaço de memória ocupado pela matriz também exige a construção de um laço, pois temos de liberar cada linha antes de liberar o vetor de ponteiros:

```
...
for (i=0; i<m; i++)
  free(mat[i]);
free(mat);
```

Operações com matrizes

Para exemplificar o uso de matrizes dinâmicas, vamos considerar a implementação de uma função que, dada uma matriz, crie dinamicamente a matriz transpos-

ta correspondente,[3] com uso das estratégias para representação de matrizes dinâmicas já discutidas.

Matriz com vetor simples

Com base na estratégia de representar a matriz com um vetor simples, podemos considerar que o protótipo da função para criar a matriz transposta é dado por:

```
float* transposta (int m, int n, float* mat);
```

onde `m` e `n` representam, respectivamente, o número de linhas e colunas da matriz `mat`, cuja transposta queremos criar. A função tem como valor de retorno o ponteiro do vetor que representa a matriz transposta criada. A implementação dessa função pode ser dada por:

```
float* transposta (int m, int n, float* mat)
{
  int i, j;
  float* trp;

  /* aloca matriz transposta */
  trp = (float*) malloc(n*m*sizeof(float));

  /* preenche matriz */
  for (i=0; i<m; i++)
     for (j=0; j<n; j++)
        trp[j*m+i] = mat[i*n+j];

  return trp;
}
```

Matriz com vetor de ponteiros

Esse mesmo problema pode ser resolvido com a estratégia de alocar a matriz por meio de um vetor de ponteiros. Nesse caso, o protótipo da função tem de mudar ligeiramente, pois a matriz passa a ser representada por um vetor de ponteiros:

```
float** transposta (int m, int n, float** mat);
```

Uma implementação para essa estratégia é mostrada a seguir. Devemos notar que, nesse caso, a complexidade adicional na alocação da matriz nos permitiu

[3] Uma matriz Q é a matriz transposta de M, se $Q_{ij} = M_{ji}$, para qualquer elemento da matriz.

acessar e atribuir os elementos a partir da sintaxe convencional de acesso a conjuntos bidimensionais.

```
float** transposta (int m, int n, float** mat)
{
  int i, j;
  float** trp;

  /* aloca matriz transposta: n linhas, m colunas */
  trp = (float**) malloc(n*sizeof(float*));
  for (i=0; i<n; i++)
     trp[i] = (float*) malloc(m*sizeof(float));

  /* preenche matriz */
  for (i=0; i<m; i++)
     for (j=0; j<n; j++)
        trp[j][i] = mat[i][j];

  return trp;
}
```

Representação de matrizes simétricas

Em uma matriz simétrica n por n, não há necessidade, no caso de i≠j, de armazenar os elementos mat[i][j] e mat[j][i], porque os dois têm o mesmo valor. Portanto, basta guardar os valores dos elementos da diagonal e de metade dos elementos restantes – por exemplo, os elementos abaixo da diagonal, para os quais i>j. Ou seja, podemos fazer uma economia de espaço usado para alocar a matriz. Em vez de n^2 valores, podemos armazenar apenas s elementos, sendo s dado por:

$$s = n + \frac{(n^2 - n)}{2} = \frac{n(n+1)}{2}$$

Podemos também determinar s como sendo a soma de uma progressão aritmética, pois temos de armazenar um elemento da primeira linha, dois elementos da segunda, três da terceira e assim por diante.

$$s = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

A representação de matrizes com essa economia de memória também pode ser feita com um vetor simples ou um vetor de ponteiros. A seguir, discutiremos a implementação de duas funções: uma para criar uma matriz quadrada simétrica e outra para acessar os elementos de uma matriz já criada.

Matriz simétrica com vetor simples

A função para criar a matriz dinamicamente usando um vetor simples não apresenta nenhuma dificuldade, pois basta dimensionar o vetor com apenas s elementos. Uma função para realizar essa tarefa é mostrada a seguir. Note que a matriz é obrigatoriamente quadrada e, portanto, só precisamos passar uma dimensão.

```
float* cria (int n)
{
  int s = n*(n+1)/2;
  float* mat = (float*) malloc(s*sizeof(float));
  return mat;
}
```

O acesso aos elementos da matriz deve ser feito como se estivéssemos representando a matriz inteira. Se for um acesso a um elemento acima da diagonal (i<j), o valor de retorno é o elemento simétrico da parte inferior, que está devidamente representado. Dessa forma, isolamos dentro do código que manipula a matriz diretamente o fato de a matriz não estar explicitamente toda armazenada. Com o uso dessa função de acesso, podemos escrever outras funções que operem sobre matrizes simétricas sem qualquer preocupação com a forma de representação interna dos elementos.

O endereçamento de um elemento da parte inferior da matriz é feito saltando-se os elementos das linhas superiores. Assim, se desejarmos acessar um elemento da quinta linha (i=4), devemos saltar 1+2+3+4 elementos, isto é, devemos saltar 1+2+...+i elementos, ou seja, i*(i+1)/2 elementos. Depois, usamos o índice j para acessar a coluna.

Como estamos projetando uma função que acessa os elementos da matriz, podemos fazer um teste adicional para evitar acessos inválidos: verificar se os índices realmente representam elementos da matriz. A função que acessa um elemento da matriz é dada a seguir.

```
float acessa (int n, float* mat, int i, int j)
{
  int k;    /* índice do elemento no vetor */

  if (i<0 || i>=n || j<0 || j>=n) {
     printf("Acesso inválido!\n");
     exit(1);
  }
  if (i>=j)
     k = i*(i+1)/2 + j;   /* acessa elemento representado */
  else
     k = j*(j+1)/2 + i;   /* acessa elemento simétrico */
  return mat[k];
}
```

Matriz simétrica com vetor de ponteiros

A estratégia de trabalhar com vetores de ponteiros para matrizes alocadas dinamicamente é muito adequada para a representação matrizes simétricas. Conforme já discutido, para otimizar o uso da memória, armazenamos apenas a parte triangular inferior da matriz. Isso significa que a primeira linha será representada por um vetor de um único elemento, a segunda linha será representada por um vetor de dois elementos e assim por diante. Como o uso de um vetor de ponteiros trata as linhas como vetores independentes, a adaptação dessa estratégia para matrizes simétricas fica simples.

Para criar a matriz, basta alocar um número variável de elementos para cada linha. O código a seguir ilustra uma possível implementação:

```
float** cria (int n)
{
  int i;
  float** mat = (float**) malloc(n*sizeof(float*));
  for (i=0; i<n; i++)
     mat[i] = (float*) malloc((i+1)*sizeof(float));
  return mat;
}
```

O acesso aos elementos é natural, se tivermos o cuidado de não acessar diretamente elementos que não estejam explicitamente alocados (isto é, elementos com `i<j`).

```
float acessa (int n, float** mat, int i, int j)
{
  if (i<0 || i>=n || j<0 || j>=n) {
     printf("Acesso inválido!\n");
     exit(1);
  }
  if (i>=j)
     return mat[i][j];    /* acessa elemento representado */
  else
     return mat[j][i];    /* acessa elemento simétrico */
}
```

Finalmente, observamos que exatamente as mesmas técnicas poderiam ser usadas para representar uma matriz "triangular", isto é, uma matriz cujos elementos acima (ou abaixo) da diagonal são todos nulos. Nesse caso, a principal diferença seria na função `acessa`, que teria como resultado o valor zero em um dos lados da diagonal, em vez de acessar o valor simétrico.

7

Cadeias de caracteres

Um texto é representado por uma seqüência (ou cadeia) de caracteres. A representação de cadeias de caracteres é de fundamental importância para o desenvolvimento de programas computacionais. Por exemplo, quando enviamos uma mensagem por correio eletrônico (e-mail), a mensagem tem de ser representada internamente no programa de mensagens para então ser enviada. De modo análogo, quando escrevemos um texto, o editor de textos é responsável por representar internamente o texto escrito, para então poder salvá-lo em disco, imprimi-lo etc. Outro exemplo importante consiste nos programas de cadastro: de clientes de um banco, de alunos em uma disciplina, de produtos de um estoque etc. Dentre os dados armazenados em cadastros, muitos são representados textualmente, como nome, endereço e descrição.

Neste capítulo, apresentaremos a forma básica para representar cadeias de caracteres em C. Antes, no entanto, temos de discutir como cada caractere é representado na linguagem.

Caracteres

Efetivamente, a linguagem C não oferece um tipo caractere. Como já discutimos, os caracteres são representados internamente na memória do computador por códigos numéricos. A linguagem C oferece então o tipo `char`, que pode armazenar valores inteiros "pequenos": um `char` tem tamanho de 1 byte, 8 bits, e pode representar assim 256 valores distintos. Como os códigos associados aos caracteres estão dentro desse intervalo, usamos o tipo `char` para representar caracteres[1]. A correspondência entre os caracteres e seus códigos numéricos é feita por uma

[1] Alguns alfabetos precisam de maior representatividade. O alfabeto chinês, por exemplo, tem mais de 256 caracteres, não sendo suficiente o tipo `char` (alguns compiladores oferecem o tipo `wchar` para esses casos).

tabela de códigos. Em geral, usa-se a tabela ASCII, mas diferentes máquinas podem usar diferentes códigos. Dessa forma, se desejamos escrever códigos portáteis, isto é, que possam ser compilados e executados em máquinas diferentes, devemos evitar o uso explícito dos códigos referentes a uma determinada tabela, como será discutido nos exemplos subseqüentes.

Como ilustração, mostramos nas Figuras 7.1 e 7.2 os códigos associados a alguns caracteres segundo a tabela ASCII.

Em C, a diferença entre caracteres e inteiros está apenas na maneira como são tratados. Por exemplo, podemos imprimir o mesmo valor de duas formas diferentes, a partir de formatos diferentes. Vamos analisar o fragmento de código abaixo:

```
char c = 97;
printf("%d %c\n",c,c);
```

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Figura 7.1 *Códigos ASCII de alguns caracteres que podem ser impressos (sp representa espaço).*

0	nul	*null*: nulo
7	bel	*bell*: campainha
8	bs	*backspace*: volta e apaga um caractere
9	ht	*tab*: tabulação horizontal
10	nl	*newline* ou *line feed*: muda de linha
13	cr	*carriage return*: volta ao início da linha
127	del	*delete*: apaga um caractere

Figura 7.2 *Códigos ASCII de alguns caracteres de controle.*

Ao se considerar a codificação de caracteres pela tabela ASCII, a variável c, que foi inicializada com o valor 97, representa o caractere a. A função printf imprime o conteúdo da variável c em dois formatos distintos: com o especificador de formato para inteiro, %d, será impresso o valor do código numérico, 97; com o formato de caractere, %c, será impresso o caractere associado ao código, isto é, a letra a.

Conforme mencionamos, devemos evitar o uso explícito de códigos de caracteres. Para tanto, a linguagem C permite a escrita de constantes caracteres. Uma constante caractere é escrita envolvendo o caractere com aspas simples. Assim, a expressão 'a' representa uma constante caractere e resulta no valor numérico associado ao caractere a. Podemos, então, reescrever o fragmento de código acima sem particularizar para a tabela ASCII.

```
char c = 'a';
printf("%d %c\n", c, c);
```

Além de agregar portabilidade e clareza ao código, o uso de constantes caracteres nos livra de ter de conhecer os códigos associados a cada caractere.

Na tabela de codificação ASCII, os dígitos são codificados em seqüência. Desse modo, se o dígito zero tem código 48, o dígito um tem obrigatoriamente código 49, e assim por diante. As letras minúsculas e as letras maiúsculas também formam dois grupos de códigos seqüenciais. Desconhecendo os códigos associados aos caracteres, podemos tirar proveito dessa codificação seqüencial para escrever programas que usam a tabela. Para exemplificar, vamos considerar a implementação de uma função para testar se um caractere c é um dígito (um dos caracteres entre '0' e '9'). Essa função pode ter como resultado 1 (verdadeiro) se c for um dígito, e 0 (falso) se não for. Uma possível implementação dessa função, que tira proveito da codificação seqüencial dos dígitos, é apresentada a seguir.

```
int digito(char c)
{
  if ((c>='0')&&(c<='9'))
     return 1;
  else
     return 0;
}
```

Da mesma maneira, podemos pensar na implementação de uma função que verifica se um determinado caractere representa uma letra. Nesse caso, basta verificar se seu código numérico representa uma letra minúscula ou maiúscula. A implementação dessa função é deixada como exercício.

Como exemplo adicional, podemos considerar uma função para converter um caractere para maiúscula. Se o caractere dado representar uma letra minúscula, devemos ter como valor de retorno a letra maiúscula correspondente. Se o ca-

ractere dado não for uma letra minúscula, devemos ter como valor de retorno o mesmo caractere, sem alteração. Uma implementação dessa função é mostrada a seguir:

```
char maiuscula(char c)
{
  /* verifica se é letra minúscula */
  if (c>= 'a' && c<='z')
     c = c-'a'+'A';   /* converte para maiúscula */
  return c;
}
```

Devemos observar que essa implementação também tira proveito da codificação seqüencial das letras minúsculas e maiúsculas na tabela de caracteres na conversão para maiúscula: se `c` é uma letra minúscula, `c-'a'` representa a "distância" entre a letra em questão e a letra `'a'`. Essa mesma distância somada ao código da letra `'A'` resulta no código da letra maiúscula correspondente.

Cadeias de caracteres (strings)

Cadeias de caracteres (strings), em C, são representadas por vetores do tipo `char` terminadas, obrigatoriamente, pelo caractere nulo (`'\0'`). Portanto, para armazenar uma cadeia de caracteres, devemos reservar uma posição adicional para o caractere de fim da cadeia. Todas as funções que manipulam cadeias de caracteres (e a biblioteca padrão de C oferece várias delas) recebem como parâmetro um vetor de `char`, isto é, um ponteiro para o primeiro elemento do vetor que representa a cadeia, e processam caractere por caractere até encontrarem o caractere nulo, o qual sinaliza o final da cadeia.

Por exemplo, o especificador de formato `%s` da função `printf` permite imprimir uma cadeia de caracteres. A função `printf` então recebe um vetor de `char` e imprime elemento por elemento até encontrar o caractere nulo. A vantagem de ter o final da cadeia delimitado pelo caractere nulo está no fato de não ser necessário passar explicitamente para as funções que recebem cadeias de caracteres o número de caracteres a ser considerado. A partir do ponteiro para o primeiro caractere, as funções processam caractere a caractere até que um `'\0'` seja encontrado.

O código a seguir ilustra a representação de uma cadeia de caracteres. Como queremos representar a palavra `Rio`, composta por 3 caracteres, declaramos um vetor com dimensão 4 (um elemento adicional para armazenarmos o caractere nulo no final da cadeia). O código preenche os elementos do vetor, incluindo o caractere `'\0'`, e imprime a palavra na tela.

```
int main ( void )
{
  char cidade[4];
  cidade[0] = 'R';
  cidade[1] = 'i';
  cidade[2] = 'o';
  cidade[3] = '\0';
  printf("%s \n", cidade);
  return 0;
}
```

Se o caractere '\0' não fosse colocado, a função printf seria executada de forma errada, pois não conseguiria identificar o final da cadeia.

Como as cadeias de caracteres são vetores, podemos reescrever o código anterior com a inicialização dos valores dos elementos do vetor na declaração:

```
int main ( void )
{
  char cidade[ ] = {'R', 'i', 'o', '\0'};
  printf("%s \n", cidade);
  return 0;
}
```

A inicialização de cadeias de caracteres é tão comum em códigos C que a linguagem permite que elas sejam inicializadas escrevendo-se os caracteres entre aspas duplas. Nesse caso, o caractere nulo é representado implicitamente. O código anterior pode ser reescrito da seguinte forma:

```
int main ( void )
{
  char cidade[ ] = "Rio";
  printf("%s \n", cidade);
  return 0;
}
```

A variável cidade é automaticamente dimensionada e inicializada com 4 elementos. Para ilustrar a declaração e a inicialização de cadeias de caracteres, consideremos as seguintes declarações:

```
char s1[ ] = "";
char s2[ ] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

Nessas declarações, a variável s1 armazena uma cadeia de caracteres vazia, representada por um vetor com um único elemento, o caractere '\0'. A variável s2 representa um vetor com 15 elementos. A variável s3 representa uma cadeia de caracteres capaz de representar cadeias com até 80 caracteres, já que foi dimen-

sionada com 81 elementos. Essa variável, no entanto, não foi inicializada e seu conteúdo é desconhecido. A variável s4 também foi dimensionada para armazenar cadeias com até 80 caracteres, mas seus primeiros quatro elementos foram atribuídos na declaração.

Leitura de caracteres e cadeias de caracteres

Para capturar o valor de um caractere simples fornecido pelo usuário via teclado, usamos a função scanf, com o especificador de formato %c.

```
char a;
...
scanf("%c", &a);
...
```

Dessa forma, se o usuário digitar a letra r, por exemplo, o código associado à letra r será armazenado na variável a. Vale ressaltar que, diferentemente dos especificadores %d e %f, o especificador %c não pula os caracteres brancos.[2] Portanto, se o usuário teclar um espaço antes da letra r, o código do espaço será capturado, e a letra r será capturada apenas em uma próxima chamada da função scanf. Se desejarmos pular todas as ocorrências de caracteres brancos que, porventura, antecedam o caractere que desejamos capturar, basta incluir um espaço em branco no formato, antes do especificador.

```
char a;
...
scanf(" %c", &a); /* o branco no formato pula brancos da entrada */
...
```

Já mencionamos que o especificador %s pode ser usado na função printf para imprimir uma cadeia de caracteres. O mesmo especificador pode ser utilizado para capturar cadeias de caracteres na função scanf. No entanto, seu uso é muito limitado. O especificador %s na função scanf pula os eventuais caracteres brancos e captura uma seqüência de caracteres não brancos. Consideremos o seguinte fragmento de código:

```
char cidade[81];
...
scanf("%s", cidade);
...
```

[2] Um "caractere branco" pode ser um espaço (' '), um caractere de tabulação ('\t') ou um caractere de nova linha ('\n').

Devemos notar que não usamos o caractere & na passagem da cadeia para a função, pois a cadeia é um vetor (o nome da variável representa o endereço do primeiro elemento do vetor, e a função atribui os valores dos elementos a partir desse endereço). O uso do especificador de formato %s na leitura é limitado, pois o fragmento de código acima funciona apenas para capturar nomes simples. Se o usuário digitar `Rio de Janeiro`, apenas a palavra `Rio` será capturada, pois o %s lê somente uma seqüência de caracteres não brancos.

Em geral, queremos ler nomes compostos (nome de pessoas, cidades, endereços para correspondência etc.). Para capturar esses nomes, podemos usar o especificador de formato `%[...]`, no qual listamos entre os colchetes todos os caracteres que aceitaremos na leitura. Assim, o formato `"%[aeiou]"` lê seqüências de vogais, isto é, a leitura prossegue até se encontrar um caractere que não seja uma vogal. Se o primeiro caractere entre colchetes for o acento circunflexo (^), teremos o efeito inverso (negação). Assim, com o formato `"%[^aeiou]"` a leitura prossegue enquanto uma vogal não for encontrada. Essa construção permite capturar nomes compostos. Consideremos o código a seguir.

```
char cidade[81];
...
scanf(" %[^\n]", cidade);
...
```

A função `scanf` agora lê uma seqüência de caracteres até que seja encontrado o caractere de mudança de linha (`'\n'`). Em termos práticos, captura-se a linha fornecida pelo usuário até que ele tecle "Enter". A inclusão do espaço no formato (antes do sinal %) garante que eventuais caracteres brancos que precedam o nome serão descartados.

Para finalizar, devemos salientar que o trecho de código citado anteriormente é perigoso, pois, se o usuário fornecer uma linha com mais de 80 caracteres, estaremos invadindo um espaço de memória não reservado (o vetor foi dimensionado com 81 elementos). Para evitar essa possível invasão, podemos limitar o número máximo de caracteres que serão capturados.

```
char cidade[81];
...
scanf(" %80[^\n]", cidade);      /* lê no máximo 80 caracteres */
...
```

Exemplos de funções que manipulam cadeias de caracteres

Nesta seção, discutiremos a implementação de algumas funções que manipulam cadeias de caracteres.

Vamos inicialmente considerar a implementação de uma função que imprime uma cadeia de caracteres, caractere por caractere. A implementação pode ser dada por:

```
void imprime (char* s)
{
  int i;
  for (i=0; s[i]!='\0'; i++)
     printf("%c",s[i]);
  printf("\n");
}
```

Devemos notar a forma como cada caractere da cadeia é acessado, até que o caractere `'\0'` seja encontrado. Esse código teria uma funcionalidade análoga à utilização do especificador de formato `%s`.

```
void imprime (char* s)
{
  printf("%s\n",s);
}
```

Consideremos agora a implementação de uma função que recebe como parâmetro de entrada uma cadeia de caracteres e fornece como retorno o número de caracteres existentes na cadeia, isto é, a função calcula o "comprimento" da cadeia. Para contar o número de caracteres da cadeia, basta contar o número de caracteres até o caractere nulo (que indica o fim da cadeia) ser encontrado. O caractere nulo em si não deve ser contado. Uma possível implementação dessa função é:

```
int comprimento (char* s)
{
  int i;
  int n = 0;  /* contador */
  for (i=0; s[i] != '\0'; i++)
     n++;
  return n;
}
```

O trecho de código a seguir faz uso dessa função.

```
#include <stdio.h>

int comprimento (char* s);

int main (void)
{
  int tam;
```

```
  char cidade[ ] = "Rio de Janeiro";
  tam = comprimento(cidade);
  printf("A string \"%s\" tem %d caracteres\n", cidade, tam);
  return 0;
}
```

A saída desse programa será: `A string "Rio de Janeiro" tem 14 caracteres.` Salientamos o uso do caractere de escape `\"` para incluir as aspas na saída.

Vamos agora considerar a implementação de uma função para copiar os elementos de uma cadeia de caracteres para outra. Conforme nossa suposição, a cadeia que receberá a cópia tem espaço suficiente para realizar a operação. A função copia os elementos da cadeia original (`orig`) para a cadeia de destino (`dest`). Uma possível implementação dessa função é mostrada a seguir:

```
void copia (char* dest, char* orig)
{
  int i;
  for (i=0; orig[i] != '\0'; i++)
     dest[i] = orig[i];
  /* fecha a cadeia copiada */
  dest[i] = '\0';
}
```

É importante ressaltar a necessidade de "fechar" a cadeia copiada após a cópia dos caracteres não nulos. Quando o laço do `for` terminar, a variável `i` terá o índice de onde está armazenado o caractere nulo na cadeia original. A cópia também deve conter o `'\0'` nessa posição.

Vamos considerar uma extensão do exemplo anterior e discutir a implementação de uma função para concatenar uma cadeia de caracteres com outra já existente. Isto é, os caracteres de uma cadeia são copiados no final da outra cadeia. Assim, se uma cadeia representa inicialmente a cadeia `PUC` e concatenarmos a ela a cadeia `Rio`, teremos como resultado a cadeia `PUCRio`. Vamos mais uma vez considerar a existência de um espaço reservado que permite fazer a cópia dos caracteres. Uma possível implementação dessa função é mostrada a seguir.

```
void concatena (char* dest, char* orig)
{
  int i = 0; /* índice usado na cadeia destino, inicializado com zero */
  int j;     /* índice usado na cadeia origem */
  /* acha o final da cadeia destino */
  i = 0;
  while (dest[i] != '\0')
     i++;
```

```
  /* copia elementos da origem para o final do destino */
  for (j=0; orig[j] != '\0'; j++)
  {
     dest[i] = orig[j];
     i++;
  }
  /* fecha cadeia destino */
  dest[i] = '\0';
}
```

Por fim, vamos considerar a implementação de uma função que compara, caractere por caractere, duas cadeias dadas. Para fazer a comparação, usaremos os códigos numéricos associados aos caracteres para determinar a ordem relativa entre eles. Dessa forma, se as duas cadeias passadas para a função forem compostas apenas por letras minúsculas ou apenas por letras maiúsculas, conseguimos determinar a ordem alfabética relativa entre elas. Para o valor de retorno da função, adotaremos a seguinte convenção: se a primeira cadeia preceder a segunda, o valor de retorno da função será -1; se a segunda preceder a primeira, será 1; se ambas as cadeias tiverem a mesma seqüência de caracteres, será 0. Uma possível implementação dessa função é mostrada a seguir:

```
int compara (char* s1, char* s2)
{
  int i;
  /* compara caractere por caractere */
  for (i=0; s1[i]!='\0' && s2[i]!='\0'; i++) {
     if (s1[i] < s2[i])
        return -1;
     else if (s1[i] > s2[i])
        return 1;
  }
  /* compara se cadeias têm o mesmo comprimento */
  if (s1[i]==s2[i])
     return 0;          /* cadeias iguais */
  else if (s2[i]!= '\0')
     return -1;         /* s1 é menor, pois tem menos caracteres */
  else
     return 1;          /* s2 é menor, pois tem menos caracteres */
}
```

Funções análogas às funções `comprimento`, `copia`, `concatena` e `compara` são disponibilizadas pela biblioteca padrão de C. As funções da biblioteca padrão são, respectivamente, `strlen`, `strcpy`, `strcat` e `strcmp`, que fazem parte da biblioteca de cadeias de caracteres, *string.h*. Existem diversas outras funções que manipulam cadeias de caracteres nessa biblioteca. A razão de mostrarmos possíveis im-

plementações dessas funções como exemplos é ilustrar a codificação da manipulação de cadeias de caracteres. Na elaboração de programas, devemos, sempre que possível, utilizar as funções da biblioteca padrão.

Consideremos agora um exemplo com alocação dinâmica. O objetivo é implementar uma função que receba como parâmetro uma cadeia de caracteres e forneça uma cópia da cadeia, alocada dinamicamente. Uma possível implementação, usando as funções da biblioteca padrão, é:

```
#include <stdlib.h>
#include <string.h>

char* duplica (char* s)
{
  int n = strlen(s);
  char* d = (char*) malloc ((n+1)*sizeof(char));
  strcpy(d,s);
  return d;
}
```

A função que chama `duplica` fica responsável por liberar o espaço alocado.

Funções recursivas

Uma cadeia de caracteres pode ser definida de forma recursiva. Podemos dizer que uma cadeia de caracteres é representada por:

- uma cadeia de caracteres vazia; ou
- um caractere seguido de uma (sub)cadeia de caracteres.

Isto é, podemos dizer que uma cadeia s não-vazia pode ser representada pelo seu primeiro caractere `s[0]` seguido da cadeia que começa no endereço do segundo caractere, `&s[1]`.

Vamos reescrever algumas das funções mostradas, agora com a versão recursiva.

Uma versão recursiva da função para imprimir a cadeia caractere por caractere é mostrada a seguir. Como já discutido, uma implementação recursiva deve ser projetada com base na definição recursiva do objeto em questão, no caso uma cadeia de caracteres. Assim, a função deve primeiro testar se a cadeia é vazia. Se for, nada precisa ser impresso; se não for, devemos imprimir o primeiro caractere e então chamar uma função para imprimir a subcadeia subseqüente. Para imprimir a subcadeia, podemos usar a própria função, recursivamente.

```
void imprime_rec (char* s)
{
  if (s[0] != '\0') {
     printf("%c",s[0]);
     imprime_rec(&s[1]);
  }
}
```

Algumas implementações ficam bem mais simples se feitas recursivamente. Por exemplo, é simples alterar a função anterior e fazer com que os caracteres da cadeia sejam impressos em ordem inversa, de trás para a frente: basta imprimir a subcadeia antes de imprimir o primeiro caractere.

```
void imprime_inv (char* s)
{
  if (s[0] != '\0') {
     imprime_inv(&s[1]);
     printf("%c",s[0]);
  }
}
```

Como exercício, sugerimos implementar a impressão inversa sem usar recursividade.

Uma implementação recursiva da função que retorna o número de caracteres existentes na cadeia é mostrada a seguir:

```
int comprimento_rec (char* s)
{
  if (s[0] == '\0')
     return 0;
  else
     return 1 + comprimento_rec(&s[1]);
}
```

Vamos mostrar agora uma possível implementação recursiva da função `copia` mostrada anteriormente.

```
void copia_rec (char* dest, char* orig)
{
  if (orig[0] == '\0')
     dest[0] = '\0';
  else {
     dest[0] = orig[0];
     copia_rec(&dest[1],&orig[1]);
  }
}
```

É fácil verificar que esse código pode ser escrito de um modo mais compacto:

```
void copia_rec (char* dest, char* orig)
{
  dest[0] = orig[0];
  if (orig[0] != '\0')
     copia_rec(&dest[1],&orig[1]);
}
```

Constante cadeia de caracteres

Em códigos C, com exceção da inicialização de cadeias de caracteres mostrada anteriormente, uma seqüência de caracteres delimitada por aspas duplas representa uma constante cadeia de caracteres, ou seja, uma expressão constante, cuja avaliação resulta no ponteiro para o qual a cadeia de caracteres está armazenada. Para exemplificar, vamos considerar este trecho de código:

```
#include <string.h>

int main ( void )
{
  char cidade[4];
  strcpy (cidade, "Rio" );
  printf ( "%s \n", cidade );
  return 0;
}
```

De forma ilustrativa, o que acontece é que, quando se encontra a cadeia "Rio", é alocada automaticamente uma área de memória com esta seqüência de caracteres:

```
'R', 'i', 'o', '\0'
```

e é fornecido o ponteiro para o primeiro elemento da seqüência. Assim, a função strcpy recebe dois ponteiros de cadeias: o primeiro aponta para o espaço associado à variável cidade, e o segundo aponta para a área em que está armazenada a cadeia constante Rio.

Dessa maneira, também é válido escrever:

```
int main (void)
{
  char *cidade;     /* declara um ponteiro para char */
  cidade = "Rio";  /* cidade recebe o endereço da cadeia "Rio" */
  printf ( "%s \n", cidade );
  return 0;
}
```

Existe uma diferença sutil entre estas duas declarações:

```
char s1[ ] = "Rio de Janeiro";
char* s2 = "Rio de Janeiro";
```

Na primeira, declaramos um vetor de `char` local inicializado com a cadeia de caracteres `Rio de Janeiro`, seguido do caractere nulo. A variável `s1` ocupa, portanto, 15 bytes de memória. Na segunda, declaramos um ponteiro para `char` inicializado com o endereço de uma área de memória em que a constante cadeia de caracteres `Rio de Janeiro` está armazenada. A variável `s2` ocupa 4 bytes (espaço de um ponteiro). Podemos verificar essa diferença ao imprimir os valores `sizeof(s1)` e `sizeof(s2)`. Como `s1` é um vetor local, podemos alterar o valor de seus elementos. Por exemplo, é válido escrever `s1[0]='X';` alterando o conteúdo da cadeia para `Xio de Janeiro`. No entanto, não é válido escrever `s2[0]='X';` pois estaríamos tentando alterar o conteúdo de um valor constante.

Vetor de cadeia de caracteres

Em muitas aplicações, desejamos representar um vetor de cadeia de caracteres. Por exemplo, podemos considerar uma aplicação que armazene os nomes de todos os alunos de uma turma em um vetor. Sabemos que uma cadeia de caracteres é representada por um vetor do tipo `char`. Para representar um vetor no qual cada elemento é uma cadeia de caracteres, devemos ter um conjunto bidimensional de `char`. Se assumirmos que o nome de nenhum aluno terá mais de 80 caracteres e que o número máximo de alunos numa turma é 50, podemos declarar um vetor bidimensional para armazenar os nomes dos alunos:

```
char alunos[50][81];
```

Com essa variável declarada, `alunos[i]` acessa a cadeia de caracteres com o nome do `(i+1)` -ésimo aluno da turma e, conseqüentemente, `alunos[i][j]` acessa a `(j+1)` -ésima letra do nome do `(i+1)` -ésimo aluno. Podemos então considerar uma função que imprime os nomes dos `n` alunos de uma turma dada por:

```
void imprime (int n, char alunos[ ][81])
{
  int i;
  for (i=0; i<n; i++)
     printf("%s\n", alunos[i]);
}
```

Para a representação de vetores de cadeias de caracteres, optamos, em geral, por declarar um vetor de ponteiros e alocar dinamicamente cada elemento (no

caso, uma cadeia de caracteres), isto é, utilizamos a estratégia de tratar cada linha da "matriz" de maneira independente. Dessa forma, otimizamos o uso do espaço de memória, pois não precisamos achar uma dimensão máxima para todas as cadeias do vetor, nem desperdiçamos espaço excessivo quando temos poucos nomes de alunos a serem armazenados. Cada elemento do vetor é um ponteiro. Se for preciso armazenar um nome na posição, alocamos o espaço de memória necessário para armazenar a cadeia de caracteres correspondente. Assim, nosso vetor com os nomes dos alunos pode ser declarado do seguinte modo:

```
#define MAX 50
char* alunos[MAX];
```

Para exemplificar, vamos escrever uma função que captura os nomes dos alunos de uma turma. A função inicialmente lê o número de alunos da turma (que deve ser menor ou igual a `MAX`) e captura os nomes fornecidos por linha, fazendo a alocação correspondente. Para escrever essa função, podemos pensar em uma função auxiliar que captura uma linha e fornece como retorno uma cadeia alocada dinamicamente com a linha inserida. Ao utilizar a função `duplica` que escrevemos anteriormente, podemos ter:

```
char* lelinha (void)
{
  char linha[121];        /* variável auxiliar para ler linha */
  printf("Digite um nome: ");
  scanf(" %120[^\n]",linha);
  return duplica(linha);
}
```

A função para capturar os nomes dos alunos preenche o vetor de nomes e pode ter como valor de retorno o número de nomes lidos:

```
int lenomes (char** alunos)
{
  int i;
  int n;
  do {
     printf("Digite o numero de alunos: ");
     scanf("%d",&n);
  } while (n>MAX);

  for (i=0; i<n; i++)
     alunos[i] = lelinha( );
  return n;
}
```

A função para liberar os nomes alocados na tabela pode ser implementada por:

```
void liberanomes (int n, char** alunos)
{
  int i;
  for (i=0; i<n; i++)
     free(alunos[i]);
}
```

Uma função para imprimir os nomes dos alunos pode ser dada por:

```
void imprimenomes (int n, char** alunos)
{
  int i;
  for (i=0; i<n; i++)
     printf("%s\n", alunos[i]);
}
```

Um programa que faz uso dessas funções é mostrado a seguir:

```
#define MAX 50

int main (void)
{
  char* alunos[MAX];
  int n = lenomes(alunos);
  imprimenomes(n,alunos);
  liberanomes(n,alunos);
  return 0;
}
```

Parâmetros da função main

Em todos os exemplos mostrados, temos considerado que a função principal, `main`, não recebe parâmetros. Na verdade, ela pode ser definida para receber zero ou dois parâmetros, geralmente chamados `argc` e `argv`. O parâmetro `argc` recebe o número de argumentos passados para o programa quando este é executado; por exemplo, de um comando de linha do sistema operacional. O parâmetro `argv` é um vetor de cadeias de caracteres que armazena os nomes passados como argumentos. Por exemplo, consideremos a função `main` declarada da seguinte forma:

```
int main (int argc, char** argv)
{
  ...
}
```

Consideremos ainda que um programa executável, com o nome mensagem, foi gerado a partir desse código. Se esse programa for invocado com a linha de comando:

```
> mensagem estruturas de dados
```

a variável argc receberá o valor 4, e o vetor argv será inicializado com os seguintes elementos: argv[0]="mensagem", argv[1]="estruturas", argv[2]="de" e argv[3]="dados". Isto é, o primeiro elemento armazena o próprio nome do executável, e os demais são preenchidos com os nomes passados na linha de comando. Esses parâmetros podem ser úteis para, por exemplo, passar o nome de um arquivo do qual serão capturados os dados de um programa. A manipulação de arquivos será discutida mais adiante neste livro. Por ora, mostraremos um exemplo simples que trata os dois parâmetros da função main.

```
#include <stdio.h>
int main (int argc, char** argv)
{
  int i;
  for (i=0; i<argc; i++)
     printf("%s\n", argv[i]);
  return 0;
}
```

Se esse programa tiver seu executável chamado de mensagem e for invocado com a linha de comando mostrada anteriormente, a saída será:

```
mensagem
estruturas
de
dados
```

8

Tipos estruturados

Até aqui, trabalhamos apenas com os tipos básicos disponibilizados pela linguagem C, como `char`, `int` e `float`. Para desenvolver programas mais complexos, precisamos trabalhar de uma maneira mais abstrata para representar os dados. É fácil imaginar que teremos de manipular dados compostos por diversas informações. Por exemplo, se nosso programa representa pontos no espaço bidimensional, a posição de cada ponto tem de ser representada por duas coordenadas (x e y). É desejável que a linguagem ofereça um mecanismo para agrupar as duas coordenadas em um mesmo contexto, para que seja possível tratar o ponto (com suas respectivas coordenadas) como um objeto (ou tipo) único. Em outros casos, o apelo por uma forma estruturada para agrupar informações fica ainda mais evidente. Tomemos como exemplo uma aplicação que deve representar o cadastro dos alunos matriculados em uma determinada disciplina. Os dados associados a cada aluno são vários: nome, número de matrícula, notas etc. Mais uma vez, é importante ter condições de estruturar todos os dados em um único contexto para representar cada aluno.

A linguagem C oferece mecanismos para estruturar dados complexos, nos quais as informações são compostas por diversos campos. Podemos então criar os tipos estruturados, que podem ser usados para representar informações como o ponto e o aluno mencionados acima. Como veremos, assim como podemos usar os tipos básicos e seus respectivos ponteiros na declaração de variáveis, podemos também usar os tipos estruturados, criados por nós.

Tipo estrutura

Em C, podemos definir um tipo de dado cujos campos são compostos de vários valores de tipos mais simples. Para ilustrar, vamos considerar o desenvolvimento de programas que manipulam pontos no plano cartesiano. Cada ponto pode ser

representado por suas coordenadas x e y, dadas por valores reais. Sem um mecanismo para agrupar as duas componentes, teríamos de representar cada ponto por duas variáveis independentes.

```
float x;
float y;
```

No entanto, desse modo, os dois valores ficam dissociados e, no caso de o programa manipular vários pontos, cabe ao programador não misturar a coordenada x de um ponto com a coordenada y de outro. Para facilitar o trabalho, a linguagem C oferece recursos para agrupar dados. Uma estrutura, em C, serve basicamente para agrupar diversas variáveis dentro de um único contexto. No nosso exemplo, podemos definir uma estrutura `ponto` que contenha os dois campos necessários para representar o ponto. A sintaxe para a definição de uma estrutura é esta:

```
struct ponto {
  float x;
  float y;
};
```

Dessa maneira, a estrutura `ponto` passa a ser um tipo, e podemos então declarar variáveis desse tipo. Após a definição da estrutura, a linha de código:

```
struct ponto p;
```

declara `p` como sendo uma variável do tipo `struct ponto`. Os elementos de uma estrutura podem ser acessados usando o operador de acesso "ponto" (`.`). Assim, é válido escrever:

```
p.x = 10.0;
p.y = 5.0;
```

Manipulamos os elementos de uma estrutura da mesma forma que variáveis simples. Podemos acessar seus valores, atribuir-lhes novos valores, acessar seus endereços etc.

Para exemplificar o uso de estruturas em programas, vamos considerar um exemplo simples em que capturamos e imprimimos as coordenadas de um ponto qualquer.

```
/* Captura e imprime as coordenadas de um ponto qualquer */

#include <stdio.h>

struct ponto {
  float x;
  float y;
};

int main (void)
{
  struct ponto p;

  printf("Digite as coordenadas do ponto(x y): ");
  scanf("%f %f", &p.x, &p.y);
  printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
  return 0;
}
```

A variável `p`, definida dentro de `main`, é uma variável local como outra qualquer. Quando a declaração é encontrada, aloca-se, na pilha de execução, um espaço para seu armazenamento, isto é, um espaço suficiente para armazenar todos os campos da estrutura (no caso, dois números reais). Notamos que o acesso ao endereço de um campo da estrutura é feito da mesma forma que com variáveis simples: basta escrever `&(p.x)`, ou simplesmente `&p.x`, pois o operador de acesso ao campo da estrutura tem precedência sobre o operador "endereço de".

Ponteiro para estruturas

Do mesmo modo que podemos declarar variáveis do tipo estrutura:

```
struct ponto p;
```

podemos também declarar variáveis do tipo ponteiro para estrutura:

```
struct ponto *pp;
```

Se a variável `pp` armazenar o endereço de uma estrutura, podemos acessar os campos dessa estrutura indiretamente, por meio de seu ponteiro:

```
(*pp).x = 12.0;
```

Nesse caso, os parênteses são indispensáveis, pois o operador "conteúdo de" tem precedência menor do que o operador de acesso. O acesso a campos de estruturas é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura. Esse operador é composto por um traço seguido de um sinal de maior, formando uma seta (->). Portanto, podemos reescrever a atribuição anterior da seguinte maneira:

```
/* Captura e imprime as coordenadas de um ponto qualquer */

#include <stdio.h>

struct ponto {
  float x;
  float y;
};

int main (void)
{
  struct ponto p;

  printf("Digite as coordenadas do ponto(x y): ");
  scanf("%f %f", &p.x, &p.y);
  printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
  return 0;
}
```

A variável p, definida dentro de main, é uma variável local como outra qualquer. Quando a declaração é encontrada, aloca-se, na pilha de execução, um espaço para seu armazenamento, isto é, um espaço suficiente para armazenar todos os campos da estrutura (no caso, dois números reais). Notamos que o acesso ao endereço de um campo da estrutura é feito da mesma forma que com variáveis simples: basta escrever &(p.x), ou simplesmente &p.x, pois o operador de acesso ao campo da estrutura tem precedência sobre o operador "endereço de".

Ponteiro para estruturas

Do mesmo modo que podemos declarar variáveis do tipo estrutura:

```
struct ponto p;
```

podemos também declarar variáveis do tipo ponteiro para estrutura:

```
struct ponto *pp;
```

Se a variável pp armazenar o endereço de uma estrutura, podemos acessar os campos dessa estrutura indiretamente, por meio de seu ponteiro:

```
(*pp).x = 12.0;
```

Nesse caso, os parênteses são indispensáveis, pois o operador "conteúdo de" tem precedência menor do que o operador de acesso. O acesso a campos de estruturas é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura. Esse operador é composto por um traço seguido de um sinal de maior, formando uma seta (->). Portanto, podemos reescrever a atribuição anterior da seguinte maneira:

```
pp->x = 12.0;
```

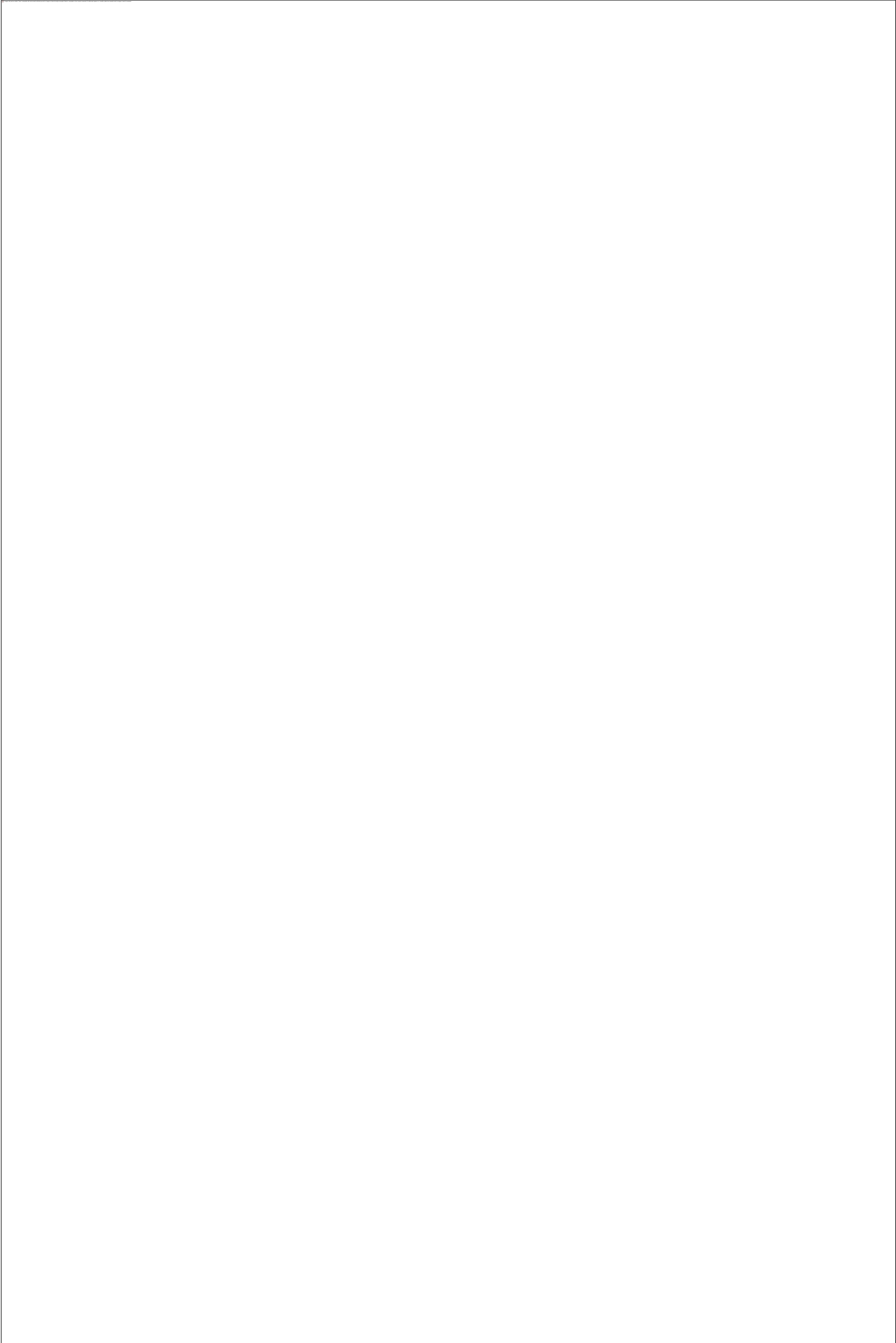
Em resumo, se temos uma variável estrutura e queremos acessar seus campos, usamos o operador de acesso ponto (p.x); se temos uma variável ponteiro para estrutura, usamos o operador de acesso seta (pp->x). Seguindo o raciocínio, se temos o ponteiro e queremos acessar o endereço de um campo, fazemos &pp->x.

Passagem de estruturas para funções

Para exemplificar a passagem de variáveis do tipo estrutura para funções, podemos reescrever o programa simples, mostrado anteriormente, que captura e imprime as coordenadas de um ponto qualquer. Inicialmente, podemos pensar em escrever uma função que imprima as coordenadas do ponto. Essa função poderia ser dada por:

```
void imprime (struct ponto p)
{
  printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
}
```

A passagem de estruturas para funções se processa de maneira análoga à passagem de variáveis simples, porém exige uma análise mais detalhada. Da forma como está escrita no código acima, a função recebe uma estrutura inteira como parâmetro. Portanto, faz-se uma cópia de toda a estrutura para a pilha, e a função acessa os dados dessa cópia. É preciso ressaltar dois pontos. Primeiro, como em toda passagem por valor, a função não tem como alterar os valores dos elementos da estrutura original (na função `imprime` isso realmente não é necessário, mas seria numa função de leitura). O segundo ponto diz respeito à eficiência, visto que copiar uma estrutura inteira para a pilha pode ser uma operação custosa (principalmente se a estrutura for muito grande). É mais conveniente passar apenas o ponteiro da estrutura, mesmo que não seja necessário alterar os valores dos elementos dentro da função, pois copiar um ponteiro para a pilha é muito mais eficiente do que copiar uma estrutura inteira. Um ponteiro ocupa em geral 4 bytes, enquanto uma estrutura pode ser definida com um tamanho arbitrariamente grande. Assim, uma segunda (e mais adequada) alternativa para escrever a função `imprime` é:

```
void imprime (struct ponto* pp)
{
  printf("O ponto fornecido foi: (%.2f,%.2f)\n", pp->x, pp->y);
}
```

Definição de "novos" tipos

A linguagem C permite criar nomes de tipos. Por exemplo, se escrevermos:

```
typedef float Real;
```

podemos usar o nome `Real` como um mnemônico para o tipo `float`. O uso de `typedef` é muito útil para abreviar nomes de tipos e para tratar tipos complexos. Alguns exemplos válidos de `typedef`:

```
typedef unsigned char UChar;
typedef int* PInt;
typedef float Vetor[4];
```

Nesse fragmento de código, definimos `UChar` como sendo o tipo `char` sem sinal, `PInt` como um tipo ponteiro para `int` e `Vetor` como um tipo que representa um vetor de quatro elementos. A partir dessas definições, podemos declarar variáveis com os mnemônicos:

```
Vetor v;
...
v[0] = 3;
...
```

Em geral, definimos nomes de tipos para as estruturas com as quais nossos programas trabalham. Por exemplo, podemos escrever:

```
struct ponto {
  float x;
  float y;
};

typedef struct ponto Ponto;
```

Assim, `Ponto` passa a representar nossa estrutura de ponto. Também podemos definir um nome para o tipo ponteiro para a estrutura.

```
typedef struct ponto *PPonto;
```

Podemos ainda definir mais de um nome num mesmo `typedef`. Os dois `typedef` anteriores poderiam ser escritos por:

```
typedef struct ponto Ponto, *PPonto;
```

Após essa definição, podemos declarar uma variável para armazenar um ponto escrevendo:

```
Ponto p;
```

De forma análoga, podemos declarar um ponteiro para um ponto assim:

```
PPonto pp;
```

Muitos programadores em C gostam de definir mnemônicos para os tipos ponteiros de estruturas (como fizemos acima para PPonto). No restante do texto, no entanto, optaremos por definir nomes apenas para as estruturas, pois consideramos que um código fica mais legível se usarmos a sintaxe da própria linguagem para a declaração de ponteiros (Ponto*).

A sintaxe de um typedef pode parecer confusa, mas é equivalente à da declaração de variáveis. Por exemplo, nesta definição:

```
typedef float Vetor[4];
```

se omitíssemos a palavra typedef, estaríamos declarando a variável Vetor como sendo um vetor de 4 elementos do tipo float. Com typedef, estamos definindo um nome que representa o tipo vetor de 4 elementos float. De maneira análoga, na definição:

```
typedef struct ponto Ponto;
```

se omitíssemos a palavra typedef, estaríamos declarando a variável Ponto como sendo do tipo struct ponto.

Por fim, vale salientar que podemos definir a estrutura e associar mnemônicos para elas em um mesmo comando:

```
typedef struct ponto {
  float x;
  float y;
} Ponto;
```

É comum os programadores de C usarem nomes com as primeiras letras maiúsculas na definição de tipos. Isso não é uma obrigatoriedade, apenas um estilo de codificação.

Aninhamento de estruturas

Os campos de uma estrutura podem ser outras estruturas previamente definidas. Para exemplificar, vamos considerar inicialmente a definição da estrutura que representa um ponto no plano (Ponto) e implementar uma função que calcula a distância entre dois pontos. Como sabemos, a distância entre dois pontos é dada por:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Uma possível implementação dessa função é mostrada a seguir. No exemplo, fazemos uso da função para cálculo da raiz quadrada (sqrt) disponibilizada pela biblioteca matemática (*math.h*). A função recebe como parâmetros os ponteiros dos pontos e tem como valor de retorno a distância correspondente.

```
float distancia (Ponto* p, Ponto* q)
{
  float d = sqrt((q->x-p->x)*(q->x-p->x) + (q->y-p->y)*(q->y-p->y));
  return d;
}
```

Agora, vamos considerar a criação de um tipo para representar círculos. Um círculo pode ser definido por seu centro (x e y) e por seu raio. Então, uma possível estrutura para a definição do círculo poderia ser:

```
struct circulo {
  float x, y;       /* centro do círculo */
  float r;          /* raio do círculo */
};
```

No entanto, se já temos o tipo Ponto definido, fica mais estruturado se definirmos o tipo Circulo usando Ponto. Ao reescrever, ficamos com:

```
struct circulo {
  Ponto p;          /* centro do círculo */
  float r;          /* raio do círculo */
};

typedef struct circulo Circulo;
```

Para ilustrar a vantagem do uso de tipos estruturados já definidos, vamos considerar a implementação de uma função que determina se um dado ponto está ou não dentro de um círculo. Essa função faz uso da função distância definida anteriormente: um ponto está dentro do círculo se sua distância ao centro do círculo for menor do que o raio.

```
int interior (Circulo* c, Ponto* p)
{
  float d = distancia(&c->p,p);
  return (d<c->r);
}
```

Devemos notar que a função, como está escrita, recebe dois ponteiros, um para o círculo e outro para o ponto que se deseja testar. Para o cálculo da distância, devemos passar para a função o endereço de dois pontos: o centro do círculo (`&c->p`) e o ponto em questão (no caso, apenas `p`, pois, dentro da função `interior`, `p` já representa o ponteiro do ponto).

Vetores de estruturas

Já discutimos o uso de vetores para agrupar elementos dos tipos básicos (vetores de inteiros, por exemplo). Nesta seção, discutiremos o uso de vetores de estruturas, isto é, vetores cujos elementos são estruturas. Para ilustrar a discussão, vamos considerar o cálculo do centro geométrico de um conjunto de pontos. Como sabemos, as coordenadas do centro geométrico são dadas por:

$$\bar{x} = \frac{\sum x_i}{n} \qquad \bar{y} = \frac{\sum y_i}{n}$$

Um vetor de estruturas pode ser usado para definir o conjunto de pontos para o qual se deseja calcular o centro geométrico. Podemos, então, escrever uma função para calcular o centro geométrico, dados o número de pontos e o vetor de pontos correspondente. A função tem como valor de retorno o ponto que representa o centro geométrico. Uma implementação dessa função é mostrada a seguir.

```
Ponto centro_geom (int n, Ponto* v)
{
  int i;
  Ponto p = {0.0f, 0.0f};  /* declara e inicializa ponto */
  for (i=0; i<n; i++)
  {
     p.x += v[i].x;
     p.y += v[i].y;
  }
  p.x /= n;
  p.y /= n;
  return p;
}
```

Devemos notar que é válido uma função ter como valor de retorno uma estrutura. No caso de estruturas pequenas (como a estrutura do ponto), esse recurso é muito útil, pois facilita o uso da função. No entanto, quando estivermos trabalhando com estruturas grandes (com muitas informações), devemos usar com critério funções que retornem valores dessas estruturas, pois a cópia do valor de retorno pode ser caro computacionalmente.

Para ilustrar um exemplo mais elaborado de vetores de estruturas, vamos considerar o cálculo da área de um polígono plano qualquer delimitado por uma seqüência de *n* pontos. A área pode ser calculada pela soma das áreas dos trapézios formados pelos lados do polígono e o eixo x, conforme ilustra a Figura 8.1.

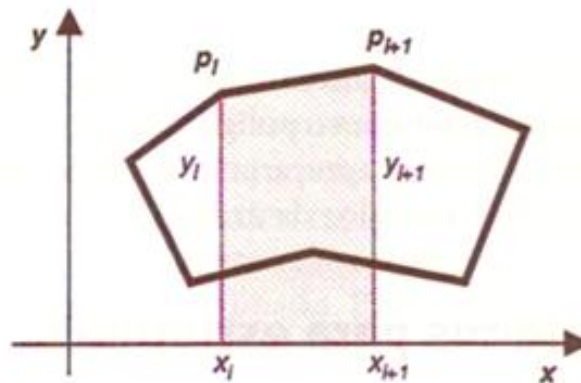



Figura 8.1 *Cálculo da área de um polígono.*

Na figura, ressaltamos a área do trapézio definido pela aresta que vai do ponto p_i ao ponto p_{i+1}. A área desse trapézio é dada por: $a = (x_{i+1} - x_i)(y_{i+1} + y_i)/2$ Quando são somadas as "áreas" (algumas delas negativas) dos trapézios definidos por todas as arestas chega-se à área do polígono (as áreas externas ao polígono são anuladas). Se a seqüência de pontos que define o polígono for dada em sentido anti-horário, chega-se a uma "área" de valor negativo. Nesse caso, a área do polígono é o valor absoluto do resultado da soma.

Um vetor de estruturas pode ser usado para definir um polígono. O polígono passa a ser representado por uma seqüência de pontos. Podemos, então, escrever uma função para calcular sua área, dados o número de pontos e o vetor de pontos que o representa. Uma implementação dessa função é mostrada a seguir:

```
float area (int n, Ponto* p)
{
  int i, j;
  float a = 0;
  for (i=0; i<n; i++) {
     j = (i+1) % n;        /* próximo índice (incremento circular) */
     a += (p[j].x-p[i].x)*(p[i].y + p[j].y)/2;
  }

  return fabs(a);
}
```

Esse código faz uso da função `fabs`, definida em *math.h*, que retorna o valor absoluto de um valor real. Um exemplo de uma função que calcula a área de um polígono é mostrado neste código:

```
int main (void)
{
 Ponto p[3] = {{1.0,1.0},{5.0,1.0},{4.0,3.0}};
 printf("area = %f\n",area (3,p));
 return 0;
}
```

Fica como exercício a tarefa de alterar esse programa para capturar do teclado o número de pontos que delimitam o polígono. O programa então alocaria dinamicamente o vetor de pontos, capturaria as coordenadas dos pontos e, chamando a função area, exibiria o valor da área.

Vetores de ponteiros para estruturas

Da mesma forma que podemos declarar vetores de estruturas, podemos também declarar vetores de ponteiros para estruturas. O uso de vetores de ponteiros é útil quando temos de tratar um conjunto de elementos complexos. Para ilustrar o uso de estruturas complexas, consideremos um exemplo em que desejamos armazenar uma tabela com dados de alunos. Podemos organizá-los em um vetor. Para cada aluno, vamos supor que sejam necessárias as seguintes informações:

- *matrícula*: número inteiro;
- *nome*: cadeia com até 80 caracteres;
- *endereço*: cadeia com até 120 caracteres;
- *telefone*: cadeia com até 20 caracteres.

Para estruturar esses dados, podemos definir um tipo que representa os dados de um aluno:

```
struct aluno {
  int mat;
  char nome[81];
  char end[121];
  char tel[21];
};

typedef struct aluno Aluno;
```

Vamos montar a tabela de alunos usando um vetor com um número máximo de alunos. Uma primeira opção é declarar um vetor de estruturas:

```
#define MAX 100
Aluno tab[MAX];
```

Dessa maneira, podemos armazenar nos elementos do vetor os dados dos alunos que queremos guardar. Seria válido, por exemplo, uma atribuição do tipo:

```
...
tab[i].mat = 9912222;
...
```

No entanto, o uso de vetores de estruturas tem, nesse caso, uma grande desvantagem. O tipo `Aluno` definido ocupa pelo menos 227 (=4+81+121+21) bytes[1]. A declaração de um vetor dessa estrutura representa um desperdício significativo de memória, pois provavelmente estaremos armazenando de fato um número de alunos bem inferior ao máximo estimado. Para contornar esse problema, podemos trabalhar com um vetor de ponteiros.

```
#define MAX 100
Aluno* tab[MAX];
```

Assim, cada elemento do vetor ocupa apenas o espaço necessário para armazenar um ponteiro. Quando precisarmos alocar os dados de um aluno em uma determinada posição do vetor, alocamos dinamicamente a estrutura `Aluno` e guardamos seu endereço no vetor de ponteiros.

Se considerarmos a utilização de um vetor de ponteiros, podemos ilustrar a implementação de algumas funcionalidades para manipular nossa tabela de alunos. Inicialmente, vamos considerar uma função de inicialização. Uma posição do vetor estará vazia, isto é, disponível para armazenar informações de um novo aluno, se o valor do seu elemento for o ponteiro nulo. Portanto, em uma função de inicialização, podemos atribuir `NULL` a todos os elementos da tabela, significando que temos, a princípio, uma tabela vazia. Devemos notar que como a função recebe um vetor de ponteiros, seu parâmetro deve ser do tipo "ponteiro para ponteiro".

```
void inicializa (int n, Aluno** tab)
{
  int i;
  for (i=0; i<n; i++)
     tab[i] = NULL;
}
```

Uma segunda funcionalidade que podemos prever armazena os dados de um novo aluno em uma posição do vetor. Vamos considerar que os dados serão for-

[1] Provavelmente o tipo ocupará um pouco mais de espaço, pois os dados têm de estar alinhados para serem armazenados na memória.

necidos via teclado e que a posição na qual os dados serão armazenados será passada para a função. Se a posição da tabela estiver vazia, devemos alocar uma nova estrutura; caso contrário, atualizamos a estrutura já apontada pelo ponteiro.

```
void preenche (int n, Aluno** tab, int i)
{
  if (i<0 || i>=n) {
     printf("Indice fora do limite do vetor\n");
     exit(1);     /* aborta o programa */
  }

  if (tab[i]==NULL)
     tab[i] = (Aluno*)malloc(sizeof(Aluno));

  printf("Entre com a matricula:");
  scanf("%d", &tab[i]->mat);
  printf("Entre com o nome:");
  scanf(" %80[^\n]", tab[i]->nome);
  printf("Entre com o endereco:");
  scanf(" %120[^\n]", tab[i]->end);
  printf("Entre com o telefone:");
  scanf(" %20[^\n]", tab[i]->tel);
}
```

Podemos também prever uma função para remover os dados de um aluno da tabela. Vamos considerar que a posição da tabela a ser liberada será passada para a função:

```
void retira (int n, Aluno** tab, int i)
{
  if (i<0 || i>=n) {
     printf("Indice fora do limite do vetor\n");
     exit(1);     /* aborta o programa */
  }

  if (tab[i] != NULL)
  {
     free(tab[i]);
     tab[i] = NULL;  /* indica que na posição não mais existe dado */
  }
}
```

Para consultar os dados, vamos considerar uma função que imprime os dados armazenados numa determinada posição do vetor:

```
void imprime (int n, Aluno** tab, int i)
{
  if (i<0 || i>=n) {
     printf("Indice fora do limite do vetor\n");
     exit(1);    /* aborta o programa */
  }

  if (tab[i] != NULL)
  {
     printf("Matrícula: %d\n", tab[i]->mat);
     printf("Nome: %s\n", tab[i]->nome);
     printf("Endereço: %s\n", tab[i]->end);
     printf("Telefone: %s\n", tab[i]->tel);
  }
}
```

Por fim, podemos implementar uma função que imprime os dados de todos os alunos da tabela:

```
void imprime_tudo (int n, Aluno** tab)
{
  int i;
  for (i=0; i<n; i++)
     imprime(i);
}
```

Um programa para testar as funções acima é mostrado a seguir. O programa declara o vetor de ponteiros, insere alguns nomes e imprime o conteúdo armazenado.

```
#include <stdio.h>

int main (void)
{
  Aluno* tab[10];
  preenche(10,tab,0);
  preenche(10,tab,1);
  preenche(10,tab,2);
  imprime_tudo(10,tab);
  retira(10,tab,0);
  retira(10,tab,1);
  retira(10,tab,2);
  return 0;
}
```

Tipo união

Em C, uma união é uma localização de memória compartilhada por diferentes variáveis, que podem ser de tipos diferentes. As uniões são usadas quando quere-

mos armazenar valores heterogêneos em um mesmo espaço de memória. A definição de uma união é parecida com a de uma estrutura:

```
union exemplo
{
  int i;
  char c;
}
```

De modo análogo à estrutura, esse fragmento de código não declara nenhuma variável, apenas define o tipo união. Após uma definição, podemos declarar variáveis do tipo união:

```
union exemplo v;
```

Na variável v, os campos i e c compartilham o mesmo espaço de memória. A variável ocupa pelo menos o espaço necessário para armazenar o maior de seus campos (um inteiro, no caso).

O acesso aos campos de uma união é análogo ao acesso a campos de uma estrutura. Usamos o operador ponto (.) para acessá-los diretamente, e o operador seta (->) para acessá-los por um ponteiro da união. Assim, dada a declaração acima, podemos escrever:

```
v.i = 10;
```

ou

```
v.c = 'x';
```

Salientamos, no entanto, que apenas um único elemento de uma união pode estar armazenado em um determinado instante, pois a atribuição a um campo da união sobrescreve o valor anteriormente atribuído a qualquer outro campo.

Tipo enumeração

Uma enumeração é um conjunto de constantes inteiras com nomes que especifica os valores legais possíveis para uma variável daquele tipo. É uma forma mais elegante de organizar valores constantes. Como exemplo, consideremos a criação de um tipo booleano. Variáveis desse tipo podem receber os valores 0 (FALSE) ou 1 (TRUE).

Poderíamos definir duas constantes simbólicas dissociadas e usar um inteiro para representar o tipo booleano:

```
#define FALSE  0
#define TRUE   1

typedef int Bool;
```

Dessa forma, as definições de FALSE e TRUE permitem a utilização desses símbolos no código, para maior clareza, mas o tipo booleano criado, como é equivalente a um inteiro qualquer, pode armazenar qualquer valor inteiro, não apenas FALSE e TRUE, o que seria mais adequado. Para validar os valores atribuídos, podemos enumerar os valores constantes que um determinado tipo pode assumir, usando enum:

```
enum bool {
  FALSE,
  TRUE
};

typedef enum bool Bool;
```

Com isso, definimos as constantes FALSE e TRUE. Por padrão, o primeiro símbolo representa o valor 0, o seguinte, o valor 1 e assim por diante. Poderíamos explicitar os valores dos símbolos em uma enumeração, por exemplo:

```
enum bool {
  TRUE = 1,
  FALSE = 0
};
```

No exemplo do tipo booleano, a numeração padrão coincide com a desejada (desde que o símbolo FALSE preceda o símbolo TRUE dentro da lista da enumeração).

A declaração de uma variável do tipo criado pode ser dada por:

```
Bool resultado;
```

onde resultado representa uma variável que pode receber apenas os valores FALSE (0) ou TRUE (1).

Exercícios

Os exercícios apresentados a seguir sugerem a implementação de diferentes funções. Para cada uma delas, o programador deve construir um programa (função `main`) para testar sua implementação.

1. Funções simples

1.1. Implemente uma função que indique se um ponto (x,y) está localizado dentro ou fora de um retângulo. O retângulo é definido por seus vértices inferior esquerdo $(x0,y0)$ e superior direito $(x1,y1)$. A função deve ter como valor de retorno 1, se o ponto estiver dentro do retângulo, e 0 caso contrário, obedecendo ao protótipo:

```
int dentro_ret (int x0, int y0, int x1, int y1, int x, int y);
```

1.2. Implemente uma função para testar se um número inteiro é primo ou não. Essa função deve obedecer ao protótipo a seguir e ter como valor de retorno 1 se `n` for primo e 0 caso contrário.

```
int primo (int n);
```

1.3. Implemente uma função que retorne o `n`-ésimo termo da série de Fibonacci. A série de Fibonacci é dada por: 1 1 2 3 5 8 13 21 ..., isto é, os dois primeiros termos são iguais a 1 e cada termo seguinte é a soma dos dois termos anteriores. Essa função deve obedecer ao protótipo:

```
int fibonacci (int n);
```

1.4. Implemente uma função que retorne a soma dos n primeiros números naturais ímpares. Essa função deve obedecer ao protótipo:

```
int soma_impares (int n);
```

1.5. Implemente uma função que retorne uma aproximação do valor de p, de acordo com a fórmula de Leibniz:

$$\pi \approx 4 * \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \dots\right)$$

Isto é:

$$\pi \approx 4 * \sum_{i=0}^{n} \frac{-1^i}{2 * i + 1}$$

Essa função deve obedecer ao seguinte protótipo, em que n indica o número de termos que deve ser usado para avaliar o valor de π:

```
double pi (int n);
```

2. Passagem de parâmetros por referência

2.1. Implemente uma função que calcule as raízes de uma equação do segundo grau, do tipo $ax^2+bx+c = 0$. Essa função deve obedecer ao protótipo:

```
int raizes (float a, float b, float c, float* x1, float* x2);
```

Essa função deve ter como valor de retorno o número de raízes reais e distintas da equação. Se existirem raízes reais, seus valores devem ser armazenados nas variáveis apontadas por x1 e x2.

2.2. Implemente uma função que calcule a área da superfície e o volume de uma esfera de raio r. Essa função deve obedecer ao protótipo:

```
void calc_esfera (float r, float* area, float* volume);
```

A área da superfície e o volume são dados, respectivamente, por $4r^2$ e $4r^3/3$.

3. Vetores

3.1. Implemente uma função que receba como parâmetro um vetor de números reais (vet) de tamanho n e retorne quantos números negativos estão armazenados nesse vetor. Essa função deve obedecer ao protótipo:

```
int negativos (int n, float* vet);
```

3.2. Implemente uma função que receba como parâmetro um vetor de números inteiros (vet) de tamanho n e retorne quantos números pares estão armazenados nesse vetor. Essa função deve obedecer ao protótipo:

```
int pares (int n, int* vet);
```

3.3. Implemente uma função que receba como parâmetro um vetor de números inteiros (vet) de tamanho n e inverta a ordem dos elementos armazenados nesse vetor. Essa função deve obedecer ao protótipo:

```
void inverte (int n, int* vet);
```

3.4. Implemente uma função que permita a avaliação de polinômios. Cada polinômio é definido por um vetor que contém seus coeficientes. Por exemplo, o polinômio de grau 2, $3x^2+2x+12$, terá um vetor de coeficientes igual a `v[ ]={12,2,3}`. A função deve obedecer ao protótipo:

```
double avalia (double* poli, int grau, double x);
```

Onde o parâmetro poli é o vetor com os coeficientes do polinômio, grau é o grau do polinômio, e x é o valor para o qual o polinômio deve ser avaliado.

3.5. Implemente uma função que calcule a derivada de um polinômio. Cada polinômio é representado como exemplificado no exercício anterior. A função deve obedecer ao protótipo:

```
void deriva(double* poli, int grau, double* out);
```

onde out é o vetor, de dimensão grau-1, no qual a função deve guardar os coeficientes do polinômio resultante da derivada.

4. Matrizes

4.1. Implemente duas versões de uma função, seguindo as diferentes estratégias discutidas para alocar matrizes, que determine se uma matriz é simétrica quadrada ou não.

4.2. Implemente um TAD, minimizando o espaço de memória utilizado, para representar uma matriz triangular inferior. Nesse tipo de matriz, todos os elementos acima da diagonal têm valor zero.

4.3. Implemente um TAD, minimizando o espaço de memória utilizado, para representar uma matriz triangular superior. Em uma matriz triangular superior, todos os elementos abaixo da diagonal têm valor zero.

5. Cadeias de caracteres

5.1. Implemente uma função que receba uma string como parâmetro e retorne como resultado o número de vogais nessa string. Essa função deve obedecer ao protótipo:

```
int conta_vogais (char* str);
```

5.2. Implemente uma função que receba como parâmetro uma string e um caractere e retorne como resultado o número de ocorrências desse caractere na string. Essa função deve obedecer ao protótipo:

```
int conta_char (char* str, char c);
```

5.3. Implemente uma função que receba uma string como parâmetro e altere nela as ocorrências de caracteres maiúsculos para minúsculos. Essa função deve obedecer ao protótipo:

```
void minusculo (char* str);
```

5.4. Implemente uma função que receba uma string como parâmetro e substitua todas as letras por suas sucessoras no alfabeto. Por exemplo, a string "Casa" seria alterada para "Dbtb". Essa função deve obedecer ao protótipo:

```
void shift_string (char* str);
```

A letra `z` deve ser substituída pela letra `a` (e `Z` por `A`). Caracteres que não forem letras devem permanecer inalterados.

5.5. Implemente uma função que receba uma string como parâmetro e substitua as ocorrências de uma letra pelo seu oposto no alfabeto, isto é, `a`↔`z`, `b`↔`y`, `c`↔`x` etc. Caracteres que não forem letras devem permanecer inalterados. Essa função deve obedecer ao protótipo:

```
void string_oposta (char* str);
```

5.6. Implemente uma função que receba uma string como parâmetro e desloque os seus caracteres uma posição para a direita. Por exemplo, a string "casa" seria alterada para "acas". Repare que o último caractere vai para o início da string. Essa função deve obedecer ao protótipo:

```
void roda_string (char* str);
```

5.7. Reimplemente as funções dos Exercícios 5.3 a 5.6 para que retornem uma nova string, alocada dentro da função, com o resultado esperado, preser-

vando as strings originais inalteradas. Essas funções devem obedecer ao seguinte protótipo:

```
char* nome_da_funcao (char* str);
```

6. Tipos estruturados

6.1. Considere uma estrutura para representar um ponto no espaço 2D e implemente uma função que indique se um dado ponto `p` está localizado dentro ou fora de um retângulo. O retângulo é definido por seus vértices inferior esquerdo `v1` e superior direito `v2`. A função deve retornar 1 caso o ponto esteja localizado dentro do retângulo, e 0 caso contrário. Essa função deve obedecer ao protótipo:

```
int dentroRet (Ponto* v1, Ponto* v2, Ponto* p);
```

6.2. Considere uma estrutura para representar um vetor no espaço 3D e implemente uma função que calcule o produto escalar de dois vetores. Essa função deve obedecer ao protótipo:

```
float dot (Vetor* v1, Vetor* v2);
```

6.3. Considere as declarações a seguir para representar o cadastro de alunos de uma disciplina e implemente uma função que imprima o número de matrícula, o nome, a turma e a média de todos os alunos aprovados na disciplina.

```
struct aluno {
  char nome[81];
  char matricula[8];
  char turma;
  float p1;
  float p2;
  float p3;
};
typedef struct aluno Aluno;
```

Assuma que o critério para aprovação é dado pela média das três provas (`p1`, `p2` e `p3`). A função recebe como parâmetros o número de alunos e um vetor de ponteiros para os dados dos alunos. Essa função deve obedecer ao protótipo:

```
void imprime_aprovados (int n, Aluno** turmas);
```

6.4. Considere as declarações do tipo `Aluno` do exercício anterior e implemente uma função que tenha como valor de retorno a média final obtida pelos alunos de uma determinada turma. A nota final de cada aluno é dada pela média das três provas.

```
float media_turma (int n, Aluno** turmas, char turma);
```

PARTE II

Estruturas dinâmicas

O conhecimento de linguagens de programação, por si só, não capacita programadores – é necessário saber usá-las de maneira eficiente. O projeto de um programa engloba, entre outras, a fase de identificação das propriedades dos dados e suas características funcionais. Uma representação adequada dos dados, em vista das funcionalidades que devem ser atendidas, constitui uma etapa fundamental para a obtenção de programas eficientes e confiáveis.

Nesta segunda parte do livro, apresentamos as estruturas de dados que convencionamos chamar de dinâmicas, pois, em geral, oferecem suporte adequado para a inserção e a remoção de elementos. Para cada elemento, essas estruturas alocam dinamicamente memória para seu armazenamento, portanto não são estruturas pré-dimensionadas. O número de elementos que podemos armazenar nessas estruturas é arbitrário.

No primeiro capítulo desta parte do livro, o Capítulo 9, introduzimos a técnica de programação baseada no conceito de tipo abstrato de dados (TAD), a qual procura encapsular (esconder) de quem usa um determinado tipo a forma concreta com que o tipo foi implementado. O Capítulo 10 mostra as estruturas de listas encadeadas, amplamente utilizadas na elaboração de programas. As estruturas de listas são inicialmente abordadas por meio de tipos de dados simples, pois assim podemos concentrar a discussão na estrutura de dados em si e não nas informações armazenadas. No final do capítulo, discutimos o armazenamento de informações estruturadas em listas e estendemos a discussão para as estruturas de listas heterogêneas, isto é, listas nas quais as informações armazenadas diferem de elemento para elemento.

Listas encadeadas, assim como vetores, são amplamente usadas para implementar diversas outras estruturas de dados com semânticas próprias. Os Capítulos 11 e 12 exibem as estruturas de pilha e fila, respectivamente, e discutem suas implementações usando vetores e listas.

O Capítulo 13 descreve as estruturas de árvores apropriadas para a organização de informações de maneira hierárquica. Por fim, no Capítulo 14, são demonstradas técnicas de programação que permitem a implementação de estruturas genéricas, isto é, estruturas que podem ser usadas para armazenar qualquer tipo de dado.

9

Tipos abstratos de dados

No capítulo anterior, apresentamos a sintaxe da linguagem C para a criação e a manipulação de tipos estruturados. Neste capítulo, discutiremos uma importante técnica de programação baseada na definição de tipos estruturados, conhecida como tipos abstratos de dados (TAD). A idéia central é encapsular (esconder) de quem usa um determinado tipo a forma concreta com que ele foi implementado. Por exemplo, se criamos um tipo para representar um ponto no espaço, um cliente desse tipo usa-o de forma abstrata, com base apenas nas funcionalidades oferecidas pelo tipo. A forma com que ele foi efetivamente implementado (armazenando cada coordenada num campo ou agrupando todas num vetor) passa a ser um detalhe de implementação, que não deve afetar o uso do tipo nos mais diversos contextos. Com isso, desacoplamos a implementação do uso, facilitamos a manutenção e aumentamos o potencial de reutilização do tipo criado. Por exemplo, a implementação do tipo pode ser alterada sem afetar seu uso em outros contextos.

Veremos como a linguagem C pode ajudar na implementação de um TAD, com alguns de seus mecanismos básicos de modularização, isto é, divisão de um programa em vários arquivos-fontes.

Módulos e compilação em separado

No Capítulo 1, mencionamos que um programa em C pode ser dividido em vários arquivos-fontes (arquivos com extensão *.c*). De fato, quando desenvolvemos programas, procuramos identificar funções afins e agrupá-las por arquivo. Quando temos um arquivo com funções que representam apenas parte da implementação de um programa completo, denominamos esse arquivo de módulo. Assim, a implementação de um programa pode ser composta por um ou mais módulos.

No caso de um programa composto por vários módulos, cada um deles deve ser compilado separadamente, gerando um arquivo objeto (em geral um arquivo com extensão *.o* ou *.obj*) para cada módulo. Após a compilação de todos os módulos, uma outra ferramenta, denominada ligador, é usada para juntar todos os arquivos objeto em um único arquivo executável.

Para programas pequenos, o uso de vários módulos pode não se justificar. No entanto, para programas de médio e grande porte, a sua divisão em vários módulos é uma técnica fundamental, pois facilita a divisão de uma tarefa maior e mais complexa em tarefas menores e, provavelmente, mais fáceis de implementar e de testar. Além disso, um módulo com funções C pode ser utilizado para compor vários programas e, assim, poupar muito tempo de programação.

Para ilustrar o uso de módulos em C, vamos considerar a existência de um arquivo *str.c* que contém apenas a implementação das funções de manipulação de strings `comprimento`, `copia` e `concatena` vistas no Capítulo 7. Considere também que temos um arquivo *prog1.c* com o seguinte código:

```
#include <stdio.h>

int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);

int main (void) {
 char str[101], str1[51], str2[51];
 printf("Digite uma seqüência de caracteres: ");
 scanf(" %50[^\n]", str1);
 printf("Digite outra seqüência de caracteres: ");
 scanf(" %50[^\n]", str2);
 copia(str, str1);
 concatena(str, str2);
 printf("Comprimento da concatenação: %d\n",comprimento(str));
 return 0;
}
```

A partir desses dois arquivos-fontes, podemos gerar um programa executável compilando cada um dos arquivos separadamente e depois ligando-os em um único arquivo executável. Por exemplo, com o compilador Gnu C (gcc), utilizaríamos a seguinte seqüência de comandos para gerar o arquivo executável *prog1.exe*:

```
> gcc -c str.c
> gcc -c prog1.c
> gcc -o prog1.exe str.o prog1.o
```

O mesmo arquivo *str.c* pode ser usado para compor outros programas que queiram utilizar suas funções. Para que as funções implementadas em *str.c* pos-

sam ser usadas por um outro módulo C, ele precisa conhecer os protótipos das funções oferecidas por *str.c*. No exemplo anterior, isso foi resolvido por meio da repetição dos protótipos das funções no início do arquivo *prog1.c*. Entretanto, para módulos que ofereçam várias funções ou que queiram usar funções de muitos outros módulos, essa repetição manual pode ficar muito trabalhosa e sensível a erros. Para contornar esse problema, todo módulo de funções C costuma ter associado um arquivo que contém apenas os protótipos das funções oferecidas pelo módulo e, às vezes, os tipos de dados exportados (`typedefs`, `structs` etc). Esse arquivo de protótipos caracteriza a interface do módulo e, em geral, segue o mesmo nome do módulo ao qual está associado, só que com a extensão *.h*. Assim, poderíamos definir um arquivo *str.h* para o módulo do exemplo anterior, com o seguinte conteúdo:

```
/* Funções oferecidas pelo módulo str.c */

/* Função comprimento
** Retorna o número de caracteres da string passada como parâmetro
*/
int comprimento (char* str);

/* Função copia
** Copia os caracteres da string orig (origem) para dest (destino)
*/
void copia (char* dest, char* orig);

/* Função concatena
** Concatena a string orig (origem) na string dest (destino)
*/
void concatena (char* dest, char* orig);
```

Observe que colocamos vários comentários no arquivo *str.h*, uma prática muito comum que tem como finalidade documentar as funções oferecidas por um módulo. Esses comentários devem esclarecer qual é o comportamento esperado das funções exportadas pelo módulo, para facilitar o seu uso por outros programadores (ou pelo mesmo programador algum tempo depois da criação do módulo).

Agora, em vez de repetir manualmente os protótipos dessas funções, todo módulo que quiser usar as funções de *str.c* precisa apenas incluir o arquivo *str.h*. No exemplo anterior, o módulo *prog1.c* poderia ser simplificado da seguinte forma:

```
#include <stdio.h>
#include "str.h"

int main (void) {
 char str[101], str1[51], str2[51];
```

```
    printf("Digite uma seqüência de caracteres: ");
    scanf(" %50[^\n]", str1);
    printf("Digite outra seqüência de caracteres: ");
    scanf(" %50[^\n]", str2);
    copia(str, str1);
    concatena(str, str2);
    printf("Comprimento da concatenação: %d\n",comprimento(str));
    return 0;
}
```

Note que os arquivos de protótipos das funções da biblioteca padrão de C (que acompanham seu compilador) são incluídos da forma `#include <arquivo.h>`, enquanto os arquivos de protótipos dos nossos módulos são geralmente incluídos da forma `#include "arquivo.h"`, conforme foi discutido no Capítulo 4.

Tipo abstrato de dados

Geralmente, um módulo agrupa vários tipos e funções com funcionalidades relacionadas, caracterizando assim uma finalidade bem definida. Por exemplo, na seção anterior, vimos um módulo com funções para a manipulação de cadeias de caracteres. Nos casos em que um módulo define um novo tipo de dado e o conjunto de operações para manipular dados desse tipo, dizemos que o módulo representa um tipo abstrato de dados (TAD). Nesse contexto, abstrato significa "esquecida a forma de implementação", ou seja, um TAD é descrito pela finalidade do tipo e de suas operações, e não pela forma como está implementado.

Podemos, por exemplo, criar um TAD para representar matrizes alocadas dinamicamente. Para isso, criamos um tipo "matriz" e uma série de funções que o manipulam. Podemos pensar, por exemplo, em funções que acessem e manipulem os valores dos elementos da matriz. Se criarmos um tipo abstrato, podemos "esconder" a estratégia de implementação. Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como é implementado, o que facilita a manutenção e a reutilização de códigos.

A divisão de programas em módulos e a criação de TADs são técnicas de programação muito importantes. Nos próximos capítulos, vamos dividir nossos exemplos e programas em módulos e usar tipos abstratos de dados sempre que possível. Antes, porém, veremos alguns exemplos completos de TADs.

A interface de um TAD consiste, basicamente, na definição do nome do tipo e do conjunto de funções exportadas para sua criação e manipulação. É comum tipos distintos oferecerem operações similares. Por exemplo, é fácil imaginar que qualquer tipo abstrato oferecerá uma função para sua criação – mais precisamente, para a criação de instâncias do tipo. Para permitir o uso de tipos distintos por um único cliente (situação muito comum em aplicações reais), precederemos os nomes das funções exportadas por um prefixo que identifica a qual tipo as fun-

ções se aplicam. Por exemplo, a função para criar um tipo Ponto pode ser chamada de pto_cria, enquanto a função para criar um tipo Circulo pode se chamar circ_cria. Assim, funções para criar tipos distintos terão nomes distintos e poderão ser usadas dentro de um mesmo contexto. Se não aplicássemos essa regra, provavelmente teríamos funções de mesmo nome exportadas por tipos distintos, o que inviabilizaria a utilização dos tipos simultaneamente, pois haveria duplicação de símbolos (um mesmo nome usado para identificar duas funções distintas)[1].

Portanto, recomendamos utilizar um prefixo nos nomes das funções exportadas pelo módulo. Se optarmos por utilizar variáveis globais e funções auxiliares na implementação dos módulos, elas serão declaradas como estáticas, e serão visíveis apenas dentro do arquivo que implementa o módulo.

Exemplo 1: TAD Ponto

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no R^2. Para isso, devemos definir um tipo abstrato, denominado Ponto, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- cria: operação que cria um ponto com coordenadas x e y;
- libera: operação que libera a memória alocada por um ponto;
- acessa: operação que retorna as coordenadas de um ponto;
- atribui: operação que atribui novos valores às coordenadas de um ponto;
- distancia: operação que calcula a distância entre dois pontos.

A interface desse módulo pode ser dada pelo arquivo *ponto.h* ilustrado a seguir:

```
/* TAD: Ponto (x,y) */

/* Tipo exportado */
typedef struct ponto Ponto;


/* Funções exportadas */

/* Função cria
** Aloca e retorna um ponto com coordenadas (x,y)
*/
Ponto* pto_cria (float x, float y);
```

[1] Na linguagem C++, é possível ter funções com mesmos nomes, diferenciadas apenas pelos tipos dos parâmetros, o que é chamado de sobrecarga de funções (*function overload*).

```
/* Função libera
** Libera a memória de um ponto previamente criado.
*/
void pto_libera (Ponto* p);

/* Função acessa
** Retorna os valores das coordenadas de um ponto
*/
void pto_acessa (Ponto* p, float* x, float* y);

/* Função atribui
** Atribui novos valores às coordenadas de um ponto
*/
void pto_atribui (Ponto* p, float x, float y);

/* Função distancia
** Retorna a distância entre dois pontos
*/
float pto_distancia (Ponto* p1, Ponto* p2);
```

Note que a composição da estrutura `Ponto` (`struct ponto`) não é exportada pelo módulo, isto é, não faz parte da interface do módulo e, portanto, não é visível para outros módulos. Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos da estrutura. Os clientes do TAD só terão acesso às informações obtidas por meio das funções exportadas pelo arquivo *ponto.h*.

Se conhecermos apenas a interface do TAD, podemos criar programas que usem as funcionalidades exportadas. O arquivo que usa o TAD deve, obrigatoriamente, incluir o arquivo responsável por definir sua interface. Por exemplo:

```
#include <stdio.h>
#include "ponto.h"

int main (void)
{
  Ponto* p = pto_cria(2.0,1.0);
  Ponto* q = pto_cria(3.4,2.1);
  float d = pto_distancia(p,q);
  printf("Distancia entre pontos: %f\n",d);
  pto_libera(q);
  pto_libera(p);
  return 0;
}
```

Logicamente, precisamos ligar o arquivo com a implementação do módulo para gerar um executável. No entanto, salientamos mais uma vez que a forma da

implementação não deve alterar o uso do tipo abstrato, isto é, podemos alterar a implementação do módulo mantendo o código anterior em funcionamento sem nenhuma alteração.

Agora, mostraremos uma implementação para esse tipo abstrato de dados. O arquivo de implementação do módulo (arquivo *ponto.c*) deve sempre incluir o arquivo de interface do módulo. Isso é necessário por duas razões. Primeiro, podem existir definições na interface que são necessárias na implementação. No nosso caso, por exemplo, precisamos da definição do tipo `Ponto`. A segunda razão é garantir que as funções implementadas correspondem às funções da interface. Como o protótipo das funções exportadas é incluído, o compilador verifica, por exemplo, se os parâmetros das funções implementadas equivalem aos parâmetros dos protótipos. Além da própria interface, precisamos naturalmente incluir as interfaces das funções usadas da biblioteca padrão.

```
#include <stdlib.h>      /* malloc, free, exit */
#include <stdio.h>       /* printf */
#include <math.h>        /* sqrt */
#include "ponto.h"
```

Como só precisamos guardar as coordenadas de um ponto, podemos definir a estrutura `ponto` da seguinte forma:

```
struct ponto {
  float x;
  float y;
};
```

A função que cria um ponto dinamicamente deve alocar a estrutura que representa o ponto e inicializar os seus campos:

```
Ponto* pto_cria (float x, float y)
{
  Ponto* p = (Ponto*) malloc(sizeof(Ponto));
  if (p == NULL) {
     printf("Memória insuficiente!\n");
     exit(1);
  }
  p->x = x;
  p->y = y;
  return p;
}
```

Para esse TAD, a função que libera um ponto deve apenas liberar a estrutura criada dinamicamente com a função `cria`:

```
void pto_libera (Ponto* p)
{
  free(p);
}
```

As funções para acessar e atribuir valores às coordenadas de um ponto são de fácil implementação, como pode ser visto a seguir. Essas funções permitem a uma função cliente acesso às coordenadas do ponto, sem conhecer a forma concreta pela qual esses valores são armazenados na estrutura que representa o tipo. Uma possível implementação dessas funções é:

```
void pto_acessa (Ponto* p, float* x, float* y)
{
  *x = p->x;
  *y = p->y;
}

void pto_atribui (Ponto* p, float x, float y)
{
  p->x = x;
  p->y = y;
}
```

Já a operação para calcular a distância entre dois pontos pode ser implementada da seguinte forma:

```
float pto_distancia (Ponto* p1, Ponto* p2)
{
  float dx = p2->x - p1->x;
  float dy = p2->y - p1->y;
  return sqrt(dx*dx + dy*dy);
}
```

Exemplo 2: TAD Círculo

Podemos aproveitar o tipo estruturado que representa um círculo do capítulo anterior e implementar um tipo abstrato de dado. As seguintes operações podem ser oferecidas:

- `cria`: operação que cria um círculo com centro (`x`,`y`)e raio `r`;
- `libera`: operação que libera a memória alocada por um círculo;
- `area`: operação que calcula a área do círculo;
- `interior`: operação que verifica se um dado ponto está dentro do círculo.

A interface desse TAD pode ser dada pelo arquivo *circulo.h* apresentado em seguida. Nos próximos exemplos, omitiremos os comentários do arquivo de interface para que o texto fique mais conciso; no entanto, em aplicações reais, recomendamos a inclusão de uma documentação adequada, na forma de comentários, nos arquivos de interface.

```
/* TAD: Círculo */

/* Dependência de módulos */
#include "ponto.h"

/* Tipo exportado */
typedef struct circulo Circulo;

/* Funções exportadas */
/* Função cria
** Aloca e retorna um círculo com centro (x,y) e raio r
*/
Circulo* circ_cria (float x, float y, float r);

/* Função libera
** Libera a memória de um círculo previamente criado.
*/
void circ_libera (Circulo* c);

/* Função area
** Retorna o valor da área do círculo.
*/
float circ_area (Circulo* c);

/* Função interior
** Verifica se um dado ponto p está dentro do círculo.
*/
int circ_interior (Circulo* c, Ponto* p);
```

Devemos notar que a operação `interior` faz uso do tipo `Ponto`, portanto a interface *ponto.h* foi incluída na interface do tipo `Circulo`.

Uma possível implementação desse tipo, arquivo `circulo.c`, é apresentada a seguir. Salientamos a existência de um TAD ponto na representação do círculo.

```
#include <stdlib.h>
#include "circulo.h"

#define PI 3.14159
```

```
struct circulo {
  Ponto* p;
  float r;
};

Circulo* circ_cria (float x, float y, float r)
{
  Circulo* c = (Circulo*)malloc(sizeof(Circulo));
  c->p = pto_cria(x,y);
  c->r = r;
  return c;
}

void circ_libera (Circulo* c)
{
  pto_libera(c->p);
  free(c);
}

float circ_area (Circulo* c)
{
  return PI*c->r*c->r;
}

int circ_interior (Circulo* c, Ponto* p)
{
  float d = pto_distancia(c->p,p);
  return (d<c->r);
}
```

Exemplo 3: TAD Matriz

Como a implementação de um TAD fica "escondida" dentro de seu módulo, podemos experimentar diferentes maneiras de implementar um mesmo TAD, sem que isso afete os clientes. Para ilustrar essa independência de implementação, vamos considerar a criação de um tipo abstrato de dados para representar matrizes de valores reais alocadas dinamicamente, com dimensões `m` por `n` fornecidas em tempo de execução. Para tanto, devemos definir um tipo abstrato, denominado `Matriz`, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- `cria`: operação que cria uma matriz de dimensão `m` por `n`;
- `libera`: operação que libera a memória alocada para a matriz;
- `acessa`: operação que acessa o elemento da linha `i` e da coluna `j` da matriz;
- `atribui`: operação que atribui o elemento da linha `i` e da coluna `j` da matriz;
- `linhas`: operação que retorna o número de linhas da matriz;
- `colunas`: operação que retorna o número de colunas da matriz.

A interface do módulo, arquivo *matriz.h*, pode ser dada por este código:

```
/* TAD: matriz m por n */

typedef struct matriz Matriz;

Matriz* mat_cria (int m, int n);
void mat_libera (Matriz* mat);
float mat_acessa (Matriz* mat, int i, int j);
void mat_atribui (Matriz* mat, int i, int j, float v);
int mat_linhas (Matriz* mat);
int mat_colunas (Matriz* mat);
```

Como discutimos no Capítulo 6, a implementação de uma matriz alocada dinamicamente pode ser feita por duas estratégias distintas: matrizes dinâmicas representadas por vetores simples e matrizes dinâmicas representadas por vetores de ponteiros. A interface do módulo independe da estratégia de implementação adotada, fato altamente desejável, pois podemos mudar a implementação sem afetar as aplicações que fazem uso do tipo abstrato. Se usarmos a estratégia com vetores simples, a estrutura que representa a matriz pode ser definida por:

```
struct matriz {
  int lin;
  int col;
  float* v;
};
```

Se usarmos a estratégia com vetores de ponteiros, a estrutura pode ser dada por:

```
struct matriz {
  int lin;
  int col;
  float** v;
};
```

Fica como exercício a implementação das funções a partir das duas estratégias alternativas. Independente da estratégia utilizada, a funcionalidade oferecida pelo tipo abstrato não se altera.

10

Listas encadeadas

Para representar um conjunto de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos apresentados agora, vamos supor que temos de desenvolver uma aplicação para representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000
int vet[MAX];
```

Ao declarar um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a Figura 10.1.

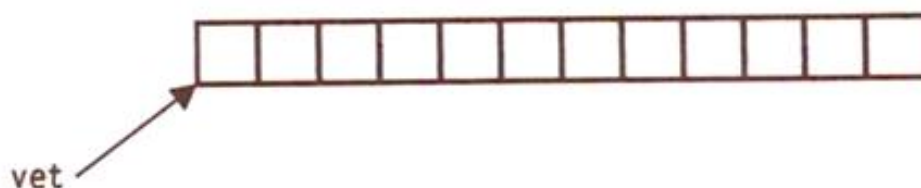



Figura 10.1 *Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.*

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo vet, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de vet é o endereço da memória em que o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor com o operador de indexação `vet[i]`. Dizemos que o vetor é uma estrutura

que possibilita o acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterar a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

A solução nesses casos consiste em utilizar estruturas de dados que cresçam conforme precisarmos armazenar novos elementos (e diminuam conforme precisarmos retirar elementos armazenados anteriormente). Essas estruturas são chamadas dinâmicas e armazenam cada um dos seus elementos por alocação dinâmica.

Nas seções a seguir, discutiremos a estrutura de dados conhecida como lista encadeada. As listas encadeadas são amplamente usadas para implementar diversas outras estruturas de dados com semânticas próprias, que serão tratadas nos capítulos seguintes.

Listas encadeadas

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Dessa forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo; portanto, não temos acesso direto aos elementos da lista. Para percorrer todos os elementos da lista, devemos explicitamente guardar o seu encadeamento, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A Figura 10.2 ilustra o arranjo da memória de uma lista encadeada.

Figura 10.2 *Arranjo da memória de uma lista encadeada.*

A estrutura consiste em uma seqüência encadeada de elementos, em geral chamados de nós da lista. Um nó da lista é representado por uma estrutura que contém, conceitualmente, dois campos: a informação armazenada e o ponteiro para o próximo elemento da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó). Do primeiro elemento, podemos alcançar o segun-

do, seguindo o encadeamento, e assim por diante. O último elemento da lista armazena, como próximo elemento, um ponteiro inválido, com valor NULL, e sinaliza, assim, que não existe um próximo elemento.

Para exemplificar a implementação de listas encadeadas em C, vamos considerar um exemplo simples em que queremos armazenar valores inteiros em uma lista encadeada. O nó da lista pode então ser representado pela estrutura a seguir.

```
struct lista {
  int info;
  struct lista* prox;
};

typedef struct lista Lista;
```

Devemos notar que se trata de uma estrutura auto-referenciada, pois, além do campo para armazenar a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Embora não seja essencial, é uma boa estratégia definir o tipo Lista como sinônimo de struct lista, conforme ilustrado anteriormente. O tipo Lista representa um nó da lista, e a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo Lista*).

De acordo com a definição de Lista, podemos definir as principais funções necessárias para implementar uma lista encadeada.

Função de criação

A função que cria uma lista vazia deve ter como valor de retorno uma lista sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro NULL, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é NULL. Uma possível implementação da função de criação é mostrada a seguir (usamos o prefixo lst para indicar que se trata de funções para manipular listas encadeadas):

```
/* função de criação: retorna uma lista vazia */
Lista* lst_cria (void)
{
  return NULL;
}
```

Função de inserção

Uma vez criada a lista vazia, podemos inserir nela novos elementos. Para cada elemento inserido, devemos alocar dinamicamente a memória necessária para

armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por essa razão, a função de inserção recebe como parâmetros de entrada a lista na qual será inserido o novo elemento e a informação do novo elemento e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
/* inserção no início: retorna a lista atualizada */
Lista* lst_insere (Lista* l, int i)
{
  Lista* novo = (Lista*) malloc(sizeof(Lista));
  novo->info = i;
  novo->prox = l;
  return novo;
}
```

Essa função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz ele apontar (isto é, tenha como próximo elemento) para o elemento que era o primeiro da lista. A função então tem como valor de retorno a nova lista, representada pelo ponteiro para o novo primeiro elemento. A Figura 10.3 ilustra a operação de inserção de um novo elemento no início da lista.

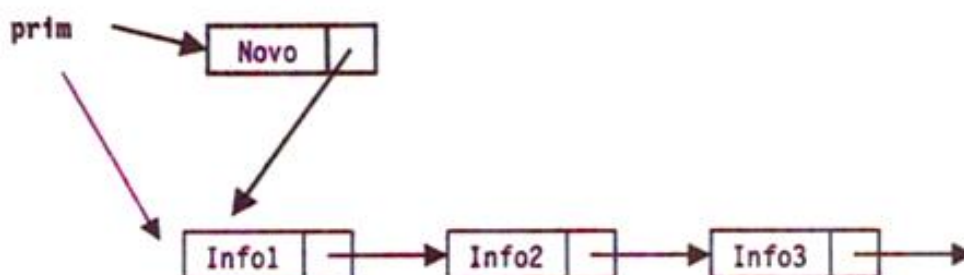



Figura 10.3 *Inserção de um novo elemento no início da lista.*

A seguir, ilustramos um trecho de código que cria uma lista inicialmente vazia e insere nela novos elementos.

```
int main (void)
{
  Lista* l;                /* declara uma lista não inicializada */
  l = lst_cria( );         /* cria e inicializa lista como vazia */
  l = lst_insere(l, 23); /* insere na lista o elemento 23 */
  l = lst_insere(l, 45); /* insere na lista o elemento 45 */
  ...
  return 0;
}
```

Observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção de um novo elemento. Se o valor de l não fosse atualizado após a inserção do primeiro elemento, estaríamos passando na segunda chamada da função insere o valor de uma lista vazia, como se o primeiro elemento não tivesse sido inserido. Como alternativa, poderíamos fazer a função insere receber o endereço da variável que representa a lista. Dessa forma, dentro da própria função insere, poderíamos atualizar o valor da variável que representa a lista na função principal. Nesse caso, os parâmetros das funções seriam do tipo ponteiro de ponteiro para lista (Lista** l), e seu conteúdo poderia ser acessado/atualizado de dentro da função por meio do operador conteúdo (*l). Uma implementação da função insere que usa essa estratégia é mostrada a seguir.

```
/* inserção no início: atualiza valor da lista */
void lst_insere (Lista** l, int i)
{
  Lista* novo = (Lista*) malloc(sizeof(Lista));
  novo->info = i;
  novo->prox = *l;
  *l = novo;
}
```

Assim, uma função cliente chamaria essa função do seguinte modo:

```
Lista* l = lst_cria( );     /* cria lista vazia */
lst_insere(&l,23);          /* insere elemento 23 */
```

A escolha de qual estratégia utilizar é uma questão de gosto do programador. A única recomendação é ser consistente, com a adoção, sempre que possível, da mesma estratégia. Para evitar o uso de ponteiro para ponteiro, sempre que possível, optaremos pela primeira versão na implementação das estruturas de dados aqui apresentadas. O uso do valor de retorno nos parece a forma mais natural de programar em C.

Função que percorre os elementos da lista

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprime os valores dos elementos armazenados em uma lista. Uma possível implementação dessa função é mostrada a seguir.

```
/* função imprime: imprime valores dos elementos */
void lst_imprime (Lista* l)
{
  Lista* p;   /* variável auxiliar para percorrer a lista */
```

```
  for (p = l; p != NULL; p = p->prox)
     printf("info = %d\n", p->info);
}
```

Recordamos que, para percorrer os elementos de um vetor, usamos uma variável auxiliar inteira a fim de armazenar os índices dos elementos. No caso da lista encadeada, a variável auxiliar tem de ser um ponteiro, usada para armazenar o endereço de cada elemento. Dentro do laço da função `imprime`, a variável `p` aponta para cada um dos elementos da lista, do primeiro até o último.

Função que verifica se a lista está vazia

Pode ser útil implementar uma função para verificar se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é `NULL`. Uma implementação dessa função é mostrada a seguir:

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int lst_vazia (Lista* l)
{
  if (l == NULL)
     return 1;
  else
     return 0;
}
```

Essa função pode ser reescrita de forma mais compacta, conforme mostrado aqui:

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int lst_vazia (Lista* l)
{
  return (l == NULL);
}
```

Função de busca

Uma outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é `NULL`.

```
/* função busca: busca um elemento na lista */
Lista* lst_busca (Lista* l, int v)
{
  Lista* p;
  for (p=l; p!=NULL; p=p->prox) {
     if (p->info == v)
        return p;
  }
  return NULL;        /* não achou o elemento */
}
```

Função que retira um elemento da lista

Devemos agora considerar a implementação de uma função que permita retirar um elemento da lista. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve atualizar o valor da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer o novo valor da lista passar a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer o elemento anterior a ele passar a apontar para o seguinte, e então podemos liberar aquele que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior a fim de acertar o encadeamento da lista. As Figuras 10.4 e 10.5 ilustram as operações de remoção.

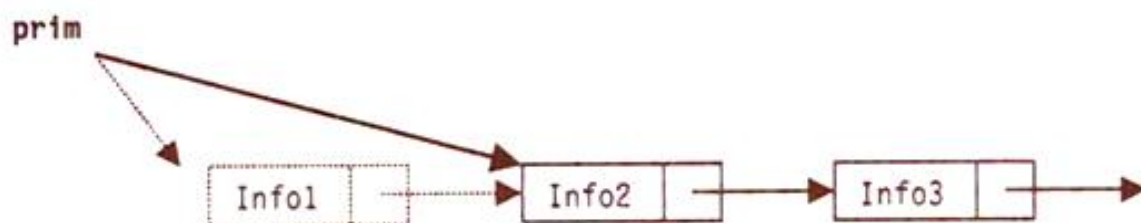



Figura 10.4 *Remoção do primeiro elemento da lista.*

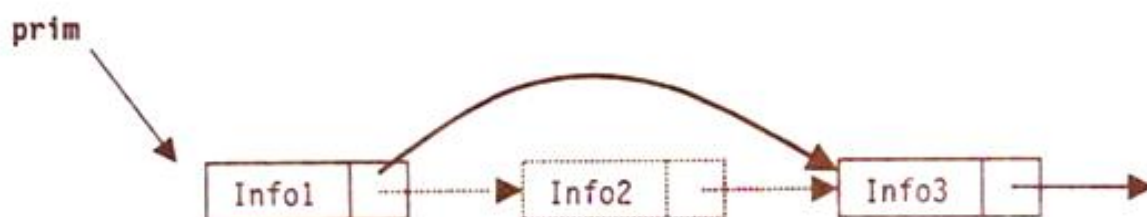



Figura 10.5 *Remoção de um elemento no meio da lista.*

Uma possível implementação da função para retirar um elemento da lista é mostrada a seguir. Inicialmente, busca-se o elemento que se deseja retirar, mas guarda-se uma referência para o elemento anterior. De modo análogo à função insere, optamos por implementar a função retira tendo como valor de retorno o eventual novo valor da lista.

```
/* função retira: retira elemento da lista */
Lista* lst_retira (Lista* l, int v)
{
  Lista* ant = NULL; /* ponteiro para elemento anterior */
  Lista* p = l;      /* ponteiro para percorrer a lista*/

  /* procura elemento na lista, guardando anterior */
  while (p != NULL && p->info != v) {
     ant = p;
     p = p->prox;
  }

  /* verifica se achou elemento */
  if (p == NULL)
     return l;   /* não achou: retorna lista original */

  /* retira elemento */
  if (ant == NULL) {
     /* retira elemento do inicio */
     l = p->prox;
  }
  else {
     /* retira elemento do meio da lista */
     ant->prox = p->prox;
  }
  free(p);
  return l;
}
```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, como pode ser observado na implementação apresentada. Mais adiante, estudaremos a implementação de filas com listas encadeadas. Em uma fila, devemos armazenar, além do ponteiro para o primeiro elemento, um ponteiro para o último elemento. Nesse caso, se for removido o último elemento, veremos que será necessário atualizar a fila.

Função para liberar a lista

Para completar o conjunto de funções básicas que manipulam uma lista, devemos considerar a função que destrói a lista, com a liberação de todos os elementos

alocados. Uma implementação dessa função é mostrada a seguir. A função percorre elemento por elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o atual (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

```
void lst_libera (Lista* l)
{
  Lista* p = l;
  while (p != NULL) {
     Lista* t = p->prox; /* guarda referência p/ próx. elemento */
     free(p);            /* libera a memória apontada por p */
     p = t;              /* faz p apontar para o próximo */
  }
}
```

TAD Lista de inteiros

Com base na implementação exemplificada, podemos criar um tipo abstrato de dados para representar uma lista encadeada de valores inteiros. A interface do módulo pode ser dada pelo arquivo *lista.h* mostrado a seguir:

```
/* TAD: lista de inteiros */

typedef struct lista Lista;

Lista* lst_cria (void);
void lst_libera (Lista* l);

Lista* lst_insere (Lista* l, int i);
Lista* lst_retira (Lista* l, int v);

int lst_vazia (Lista* l);
Lista* lst_busca (Lista* l, int v);
void lst_imprime (Lista* l);
```

A partir dessa interface, podemos criar um programa que utiliza as funções de lista exportadas.

```
#include <stdio.h>
#include "lista.h"

int main (void)
{
  Lista* l;               /* declara uma lista não iniciada */
  l = lst_cria( );         /* inicia lista vazia */
```

```
    l = lst_insere(l, 23); /* insere na lista o elemento 23 */
    l = lst_insere(l, 45); /* insere na lista o elemento 45 */
    l = lst_insere(l, 56); /* insere na lista o elemento 56 */
    l = lst_insere(l, 78); /* insere na lista o elemento 78 */
    lst_imprime(l);        /* imprimirá: 78 56 45 23 */
    l = lst_retira(l, 78);
    lst_imprime(l);        /* imprimirá: 56 45 23 */
    l = lst_retira(l, 45);
    lst_imprime(l);        /* imprimirá: 56 23 */
    lst_libera(l);
    return 0;
}
```

Mais uma vez, observe que, na função cliente (`main`, no exemplo mostrado), não podemos deixar de atualizar a variável que representa a lista a cada inserção e a cada remoção de um elemento. Esquecer de atribuir o valor de retorno à variável que representa a lista pode gerar erros graves. Se, por exemplo, a função retirar o primeiro elemento da lista, a variável que representa a lista, se não fosse atualizada, estaria apontando para um nó já liberado. Como já mencionamos, uma alternativa seria fazer as funções `insere` e `retira` receberem o endereço da variável que representa a lista.

Manutenção da lista ordenada

A função de inserção vista anteriormente armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista. Se quisermos manter os elementos na lista em uma determinada ordem, temos de encontrar a posição correta para inserir o novo elemento. Essa função não é eficiente, pois temos de percorrer a lista, elemento por elemento, para achar a posição de inserção. Se a ordem de armazenamento dos elementos dentro da lista não for relevante, optamos por fazer inserções no início, pois o custo computacional dessa operação independe do número de elementos na lista.

No entanto, se desejarmos manter os elementos em ordem, cada novo elemento deve ser inserido na ordem correta. Para exemplificar, vamos considerar que queremos manter nossa lista de números inteiros em ordem crescente. A função de inserção, nesse caso, tem a mesma assinatura da função de inserção mostrada anteriormente, mas percorre os elementos da lista a fim de encontrar a posição correta para a inserção do novo. Com isso, temos de saber inserir um elemento no meio da lista. A Figura 10.6 ilustra a inserção de um elemento no meio da lista.

Conforme ilustrado na Figura 10.6, devemos localizar o elemento da lista que precederá o elemento novo a ser inserido. De posse do ponteiro para esse elemento, podemos encadear o novo elemento na lista. Ele apontará para o próximo elemento na lista, e o elemento precedente apontará para o novo. O código a seguir ilustra a implementação dessa função.

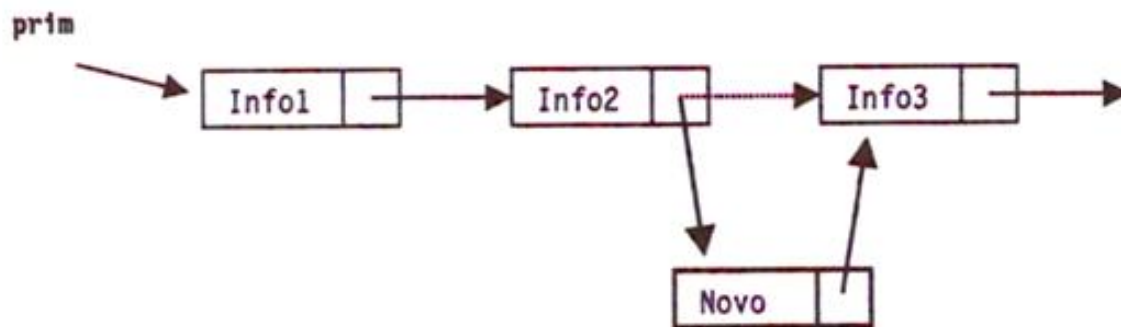



Figura 10.6 *Inserção de um elemento no meio da lista.*

```
/* função insere_ordenado: insere elemento em ordem */
Lista* lst_insere_ordenado (Lista* l, int v)
{
  Lista* novo;
  Lista* ant = NULL;      /* ponteiro para elemento anterior */
  Lista* p = l;           /* ponteiro para percorrer a lista*/

  /* procura posição de inserção */
  while (p != NULL && p->info < v) {
     ant = p;
     p = p->prox;
  }

  /* cria novo elemento */
  novo = (Lista*) malloc(sizeof(Lista));
  novo->info = v;

  /* encadeia elemento */
  if (ant == NULL) {   /* insere elemento no início */
     novo->prox = l;
     l = novo;
  }
  else {   /* insere elemento no meio da lista */
     novo->prox = ant->prox;
     ant->prox = novo;
  }
  return l;
}
```

Devemos notar que essa função, analogamente ao observado para a função de remoção, também funciona se o elemento tiver de ser inserido no final da lista.

Implementações recursivas

Uma lista pode ser definida de maneira recursiva. Podemos dizer que uma lista encadeada é representada por:

- uma lista vazia; ou
- um elemento seguido de uma (sub)lista.

Nesse último caso, o segundo elemento da lista representa o primeiro elemento da sublista. A Figura 10.7 ilustra uma representação gráfica dessa definição recursiva de lista encadeada.

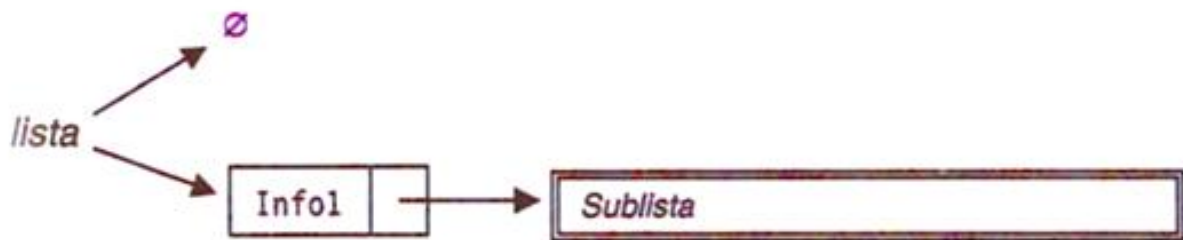



Figura 10.7 *Representação gráfica da definição recursiva.*

Com base na definição recursiva, podemos implementar as funções de lista recursivamente. Por exemplo, vamos considerar uma função para imprimir os elementos da lista. Devemos seguir a definição recursiva para implementar a função. Assim, devemos primeiramente verificar se a lista é vazia. Se for, não temos nada para imprimir. Caso contrário, a lista é composta pelo primeiro nó, dado por `l`, e por uma sublista, dada por `l->prox`. Assim, devemos imprimir a informação associada ao primeiro nó, acessando `l->info`, e imprimir as informações da sublista. Para imprimir a sublista, podemos usar a própria função que estamos codificando, pois nossa função é para imprimir qualquer lista de inteiros. Uma possível implementação dessa função é mostrada a seguir:

```
/* Função imprime recursiva */
void lst_imprime_rec (Lista* l)
{
  if (lst_vazia(l))
     return;
  else {
     /* imprime primeiro elemento */
     printf("info: %d\n",l->info);
     /* imprime sub-lista */
     lst_imprime_rec(l->prox);
  }
}
```

É fácil observar que essa mesma função pode ser reescrita de forma mais compacta invertendo o teste condicional:

```
/* Função imprime recursiva */
void lst_imprime_rec (Lista* l)
{
  if (!lst_vazia(l)) {
     /* imprime primeiro elemento */
     printf("info: %d\n",l->info);
     /* imprime sub-lista */
     lst_imprime_rec(l->prox);
  }
}
```

Não recomendamos tentar seguir, passo a passo, a execução de uma implementação recursiva e sim entendê-la com base apenas na definição recursiva do objeto em questão – no caso, a lista encadeada.

É fácil alterar esse código para obter a impressão dos elementos da lista em ordem inversa: basta inverter a ordem das chamadas às funções `printf` e `imprime_rec`.

A função para retirar um elemento da lista também pode ser escrita de forma recursiva. Nesse caso, só retiramos um elemento se ele for o primeiro da lista (ou da sublista). Se o elemento que queremos retirar não for o primeiro, chamamos a função recursivamente para retirar o elemento da sublista.

```
/* Função retira recursiva */
Lista* lst_retira_rec (Lista* l, int v)
{
  if (!lst_vazia(l)) {
     /* verifica se elemento a ser retirado é o primeiro */
     if (l->info == v) {
        Lista* t = l;     /* temporário para poder liberar */
        l = l->prox;
        free(t);
     }
     else {
        /* retira de sub-lista */
        l->prox = lst_retira_rec(l->prox,v);
     }
  }
  return l;
}
```

Salientamos apenas a necessidade de reatribuir o valor de `l->prox` na chamada recursiva, já que a função pode alterar o valor da sublista.

A função para liberar uma lista também pode ser escrita recursivamente, de forma bastante simples. Nessa função, se a lista não for vazia, liberamos primeiro a sublista e depois liberamos a lista.

```
void lst_libera_rec (Lista* l)
{
  if (!lst_vazia(l))
  {
     lst_libera_rec(l->prox);
     free(l);
  }
}
```

Função para comparar duas listas

A utilização de implementações recursivas para listas encadeadas é uma questão de opção. Em geral, a implementação não recursiva é mais eficiente do ponto de vista do esforço computacional dispensado, pois minimiza o número de chamadas de funções, que são operações relativamente caras. No entanto, algumas implementações podem ficar mais simples se feitas de forma recursiva.

Para ilustrar essa discussão, vamos considerar a implementação de uma função para testar se duas listas dadas são iguais. Duas listas são consideradas iguais se têm a mesma seqüência de elementos, naturalmente com o mesmo número de elementos. O protótipo dessa função é dado por:

```
int lst_igual (Lista* l1, Lista* l2);
```

A implementação dessa função de forma não recursiva requer que tenhamos dois ponteiros auxiliares para percorrer as duas listas, simultaneamente, e comparar as informações associadas a cada par de elementos. Se encontrarmos informações diferentes, podemos concluir que as listas são diferentes. Esse teste deve ser feito até que uma das listas (ou as duas) chegue ao fim. Fora do laço, testamos se os dois ponteiros auxiliares são iguais. Se forem, significa que ambos são `NULL`, isto é, as duas listas têm o mesmo número de elementos. Uma implementação dessa função é mostrada a seguir (essa função é análoga à função que compara duas cadeias de caracteres, apresentada no Capítulo 7).

```
int lst_igual (Lista* l1, Lista* l2)
{
  Lista* p1;   /* ponteiro para percorrer l1 */
  Lista* p2;   /* ponteiro para percorrer l2 */

  for (p1=l1,p2=l2; p1!=NULL&&p2!=NULL; p1=p1->prox,p2=p2->prox) {
     if (p1->info != p2->info)
        return 0;
  }
  return p1==p2;
}
```

Uma implementação recursiva dessa função deve ser pensada com base na definição recursiva de lista. Primeiramente, temos de testar os casos bases, nos

quais as listas podem ser vazias. Dessa forma, verificamos se as duas listas dadas são vazias. Se forem, logicamente elas são iguais. Se não forem, devemos verificar se uma delas é vazia. Se for, concluímos que se tratam de listas diferentes. Se ambas não forem vazias, devemos testar a igualdade entre as informações associadas aos primeiros nós das listas e verificar a igualdade das sublistas. Uma possível implementação é mostrada a seguir:

```
int lst_igual (Lista* l1, Lista* l2)
{
  if (l1==NULL && l2==NULL)
     return 1;
  else if (l1==NULL || l2==NULL)
     return 0;
  else
     return l1->info==l2->info && lst_igual(l1->prox,l2->prox);
}
```

Listas circulares

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, numa aplicação que manipula figuras geométricas, as arestas que delimitam uma face podem ser agrupadas por uma estrutura circular. Para esses casos, podemos usar listas circulares.

Em uma lista circular, o último elemento tem como próximo o primeiro elemento da lista, o que forma um ciclo. A rigor, nesse caso, não faz sentido falar em primeiro ou último elemento. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista. A Figura 10.8 ilustra o arranjo da memória para a representação de uma lista circular.

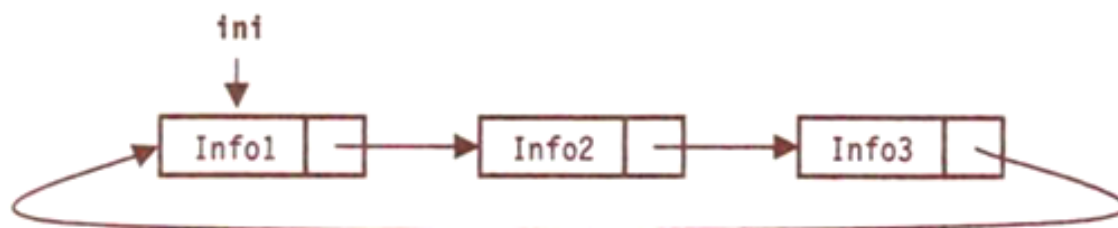



Figura 10.8 *Arranjo da memória de uma lista circular.*

Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançar novamente esse mesmo elemento. O código a seguir exemplifica essa forma de percorrer os elementos. Para simplificar, consideramos uma lista que armazena valores inteiros. Devemos salientar que o caso no qual a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um elemento inicial vale `NULL`).

```
void lcirc_imprime (Lista* l)
```

```
void lcirc_imprime (Lista* l)
{
  Lista* p = l;         /* faz p apontar para o nó inicial */
  /* testa se lista não é vazia e então percorre com do-while */
  if (p) do {
     printf("%d\n", p->info);   /* imprime informação do nó */
     p = p->prox;               /* avança para o próximo nó */
  } while (p != l);
}
```

Listas duplamente encadeadas

A estrutura de lista encadeada vista nas seções anteriores caracteriza-se por formar um encadeamento simples entre os elementos: cada elemento armazena um ponteiro para o próximo elemento da lista. Dessa forma, não temos como percorrer eficientemente os elementos em ordem inversa, isto é, do final para o início da lista. O encadeamento simples também dificulta a retirada de um elemento da lista. Mesmo se tivermos o ponteiro do elemento que desejamos retirar, temos de percorrer a lista, elemento por elemento, para encontrar o elemento anterior, pois, dado o ponteiro para um determinado elemento, não temos como acessar diretamente seu elemento anterior.

Para solucionar esses problemas, podemos formar o que chamamos de listas duplamente encadeadas. Nelas, cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Assim, dado um elemento, podemos acessar os dois elementos adjacentes: o próximo e o anterior. Se tivermos um ponteiro para o último elemento da lista, podemos percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior até alcançar o primeiro elemento da lista, que não tem um elemento anterior (seu ponteiro vale `NULL`). A Figura 10.9 esquematiza a estruturação de uma lista duplamente encadeada.

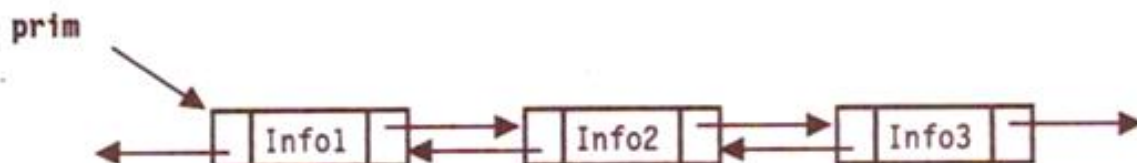



Figura 10.9 *Arranjo da memória de uma lista duplamente encadeada.*

Para exemplificar a implementação de listas duplamente encadeadas, vamos novamente considerar o exemplo simples no qual queremos armazenar valores inteiros na lista. O nó da lista pode ser representado pela estrutura a seguir, e a lista pode ser representada com o ponteiro para o primeiro nó.

```
struct lista2 {
  int info;
  struct lista2* ant;
  struct lista2* prox;
};

typedef struct lista2 Lista2;
```

Com base nessas definições, exemplificaremos a seguir a implementação de algumas funções que manipulam listas duplamente encadeadas.

Função de inserção

O código a seguir mostra uma possível implementação da função que insere novos elementos no início da lista. Após a alocação do novo elemento, a função acerta o duplo encadeamento.

```
/* inserção no início */
Lista2* lst2_insere (Lista2* l, int v)
{
  Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
  novo->info = v;
  novo->prox = l;
  novo->ant = NULL;
  /* verifica se lista não está vazia */
  if (l != NULL)
     l->ant = novo;
  return novo;
}
```

Nessa função, o novo elemento é encadeado no início da lista. Assim, ele tem como próximo elemento o antigo primeiro elemento da lista e como anterior o valor NULL. A seguir, a função testa se a lista não era vazia, pois, nesse caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento. De qualquer modo, o novo elemento passa a ser o primeiro da lista e deve ser retornado como valor da lista atualizada. A Figura 10.10 ilustra a operação de inserção de um novo elemento no início da lista.

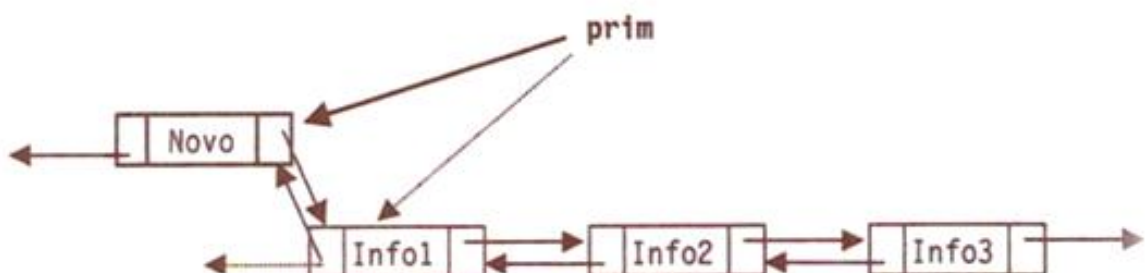



Figura 10.10 *Inserção de um novo elemento no início da lista.*

Função de busca

A função de busca recebe a informação referente ao elemento que queremos buscar e tem como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

```
/* função busca: busca um elemento na lista */
Lista2* lst2_busca (Lista2* l, int v)
{
  Lista2* p;
  for (p=l; p!=NULL; p=p->prox)
     if (p->info == v)
        return p;
  return NULL;       /* não achou o elemento */
}
```

Conforme notamos, essa função tem uma implementação igual ao caso da lista simplesmente encadeada, pois só usamos o ponteiro para o próximo elemento.

Função que retira um elemento da lista

A função de remoção fica mais complicada, pois temos de acertar o encadeamento duplo. Em contrapartida, podemos retirar um elemento da lista se conhecermos apenas o ponteiro para esse elemento. Dessa forma, podemos usar a função de busca anteriormente citada para localizar o elemento e, em seguida, acertar o encadeamento, para, então, liberar o elemento ao final.

Se p representa o ponteiro do elemento que desejamos retirar, para acertar o encadeamento devemos conceitualmente fazer:

```
p->ant->prox = p->prox;
p->prox->ant = p->ant;
```

isto é, o anterior passa a apontar para o próximo, e o próximo passa a apontar para o anterior. Quando p apontar para um elemento no meio da lista, as duas atribuições acima são suficientes para efetivamente acertar o encadeamento da lista. No entanto, se p for um elemento no extremo da lista, devemos considerar as condições de contorno. Se p for o último elemento, não podemos escrever p->prox->ant, pois p->prox é NULL. Analogamente, se p apontar para o primeiro elemento, também não podemos escrever p->ant->prox; além disso, temos de atualizar o valor da lista, pois o primeiro elemento será removido.

Uma implementação da função para retirar um elemento é mostrada a seguir:

```
/* função retira: retira elemento da lista */
Lista2* lst2_retira (Lista2* l, int v) {
  Lista2* p = busca(l,v);

  if (p == NULL)
     return l;    /* não achou o elemento: retorna lista inalterada */

  /* retira elemento do encadeamento */
  if (l == p)              /* testa se é o primeiro elemento */
     l = p->prox;
  else
     p->ant->prox = p->prox;

  if (p->prox != NULL)    /* testa se é o último elemento */
     p->prox->ant = p->ant;

  free(p);

  return l;
}
```

Lista circular duplamente encadeada

Uma lista circular também pode ser construída com encadeamento duplo. Nesse caso, o que seria o último elemento da lista passa a ter como próximo o primeiro elemento, que, por sua vez, passa a ter o último como anterior. Com essa construção, podemos percorrer a lista nos dois sentidos, a partir de um ponteiro para um elemento qualquer. A seguir, ilustramos o código para imprimir a lista no sentido reverso, isto é, percorrer o encadeamento dos elementos anteriores.

```
void l2circ_imprime_rev (Lista2* l)
{
  Lista2* p = l;         /* faz p apontar para o nó inicial */
  /* testa se lista não é vazia e então percorre com do-while */
  if (p) do {
     printf("%d\n", p->info);   /* imprime informação do nó */
     p = p->ant;                /* "avança" para o nó anterior */
  } while (p != l);
}
```

Listas de tipos estruturados

Nos exemplos anteriores, trabalhamos com informações simples, pois tínhamos como principal objetivo discutir a estrutura das listas. Logicamente, a informação associada a cada nó de uma lista encadeada pode ser mais complexa. Como

veremos, independente da informação armazenada e da forma como ela é representada internamente, o encadeamento dos elementos da lista não é alterado. As funções apresentadas para manipular listas de inteiros podem ser facilmente adaptadas para tratar listas de outros tipos. Para simplificar essa exposição, vamos discutir a representação de tipos estruturados em uma lista simplesmente encadeada. As mesmas técnicas de programação podem ser usadas em outras estruturas de listas.

Conforme mencionamos, um nó de uma lista encadeada contém basicamente dois componentes: o encadeamento e a informação armazenada. Assim, a estrutura de um nó para representar uma lista de números inteiros é dada por:

```
struct lista {
  int info;
  struct lista *prox;
};
```

Analogamente, se quisermos representar uma lista de números reais, podemos definir a estrutura do nó como:

```
struct lista {
  float info;
  struct lista *prox;
};
```

A informação armazenada na lista não precisa ser necessariamente um dado simples. Podemos, por exemplo, considerar a construção de uma lista para armazenar um conjunto de retângulos, com cada retângulo sendo definido pela base `b` e pela altura `h`. Assim, a estrutura do nó pode ser dada por:

```
struct lista {
  float b;
  float h;
  struct lista *prox;
};
```

Com isso, uma função auxiliar para alocar um nó dessa lista inicializando a informação pode ser dada por (considerando `Lista` sinônimo da `struct lista`):

```
static Lista* aloca (float b, float h)
{
  Lista* p = (Lista*)malloc(sizeof(Lista));
  p->b = b;
  p->h = h;
  return p;
}
```

Essa mesma composição de nó pode ser escrita de forma mais clara se definirmos um tipo adicional que represente a informação. Podemos definir um tipo `Retangulo` e usá-lo para representar a informação armazenada na lista.

```
struct retangulo {
  float b;
  float h;
};
typedef struct retangulo Retangulo;

struct lista {
  Retangulo info;
  struct lista *prox;
};
```

Assim, a nossa função auxiliar ficaria:

```
static Lista* aloca (float b, float h)
{
  Lista* p = (Lista*)malloc(sizeof(Lista));
  p->info.b = b;
  p->info.h = h;
  return p;
}
```

Aqui, a informação volta a ser representada por um único campo (`info`), que é uma estrutura. Ainda mais interessante é ter o campo da informação representado por um ponteiro para uma estrutura, em vez da estrutura em si.

```
struct retangulo {
  float b;
  float h;
};
typedef struct retangulo Retangulo;

struct lista {
  Retangulo *info;
  struct lista *prox;
};
typedef struct lista Lista;
```

Nesse caso, para alocar um nó, temos de fazer duas alocações dinâmicas: uma para criar a estrutura do retângulo e outra para criar a estrutura do nó. O código a seguir ilustra uma função para a alocação de um nó.

```
static Lista* aloca (float b, float h)
{
  Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
  Lista* p = (Lista*) malloc(sizeof(Lista));
  r->b = b;
  r->h = h;
  p->info = r;
  p->prox = NULL;
  return p;
}
```

Dessa maneira, o valor da base associado a um nó `p` seria acessado por: `p->info->b`. A vantagem dessa representação (a qual utiliza ponteiros) é que, independentemente da informação armazenada na lista, a estrutura do nó é sempre composta por um ponteiro para a informação e um ponteiro para o próximo nó da lista.

Listas heterogêneas

A representação da informação por um ponteiro permite construir listas heterogêneas, isto é, listas em que as informações armazenadas diferem de nó para nó. Como exemplo, vamos considerar uma aplicação que necessite manipular listas de objetos geométricos planos para cálculos de áreas. Para simplificar, vamos considerar que os objetos podem ser apenas retângulos, triângulos ou círculos. Sabemos que as áreas desses objetos são dadas por:

$$r = b * h \qquad t = \frac{b * h}{2} \qquad c = \pi\, r^2$$

Devemos definir um tipo para cada objeto a ser representado:

```
struct retangulo {
  float b;
  float h;
};
typedef struct retangulo Retangulo;

struct triangulo {
  float b;
  float h;
};
typedef struct triangulo Triangulo;

struct circulo {
  float r;
};
typedef struct circulo Circulo;
```

O nó da lista deve então ser composto por três campos:

- um identificador de qual objeto está armazenado no nó;
- um ponteiro para a estrutura que contém a informação;
- um ponteiro para o próximo nó da lista.

É importante salientar que, a rigor, a lista é homogênea, ou seja, todos os nós contêm os mesmos campos. O ponteiro para a informação deve ser do tipo genérico, pois não sabemos a princípio para que estrutura ele apontará: pode apontar para um retângulo, um triângulo ou um círculo. Um ponteiro genérico em C é representado pelo tipo `void*`. Uma variável do tipo "ponteiro genérico" pode representar qualquer endereço de memória, independente da informação de fato armazenada nesse espaço. No entanto, de posse de um ponteiro genérico, não podemos acessar a memória apontada por ele, já que não sabemos a informação armazenada. Por isso, o nó de uma lista genérica deve guardar explicitamente um identificador do tipo de objeto armazenado de fato. Consultando esse identificador, podemos converter o ponteiro genérico no ponteiro específico para o objeto em questão e, então, acessar os campos do objeto.

Como identificador de tipo, podemos usar valores inteiros definidos como constantes simbólicas:

```
#define RET 0
#define TRI 1
#define CIR 2
```

Assim, na criação do nó, armazenamos o identificador de tipo correspondente ao objeto representado. A estrutura que representa o nó pode então ser dada por:

```
/* Define o nó da estrutura */
struct listahet {
  int   tipo;
  void *info;
  struct listahet *prox;
};
typedef struct listahet ListaHet;
```

A função para a criação de um nó da lista pode ser definida por três variações, uma para cada tipo de objeto que pode ser armazenado.

```
/* Cria um nó com um retângulo */
ListaHet* cria_ret (float b, float h)
{
  Retangulo* r;
  ListaHet* p;
```

```
  /* aloca retângulo */
  r = (Retangulo*) malloc(sizeof(Retangulo));
  r->b = b;
  r->h = h;
  /* aloca nó */
  p = (ListaHet*) malloc(sizeof(ListaHet));
  p->tipo = RET;
  p->info = r;
  p->prox = NULL;
  return p;
}

/* Cria um nó com um triângulo */
ListaHet* cria_tri (float b, float h)
{
  Triangulo* t;
  ListaHet* p;
  /* aloca triângulo */
  t = (Triangulo*) malloc(sizeof(Triangulo));
  t->b = b;
  t->h = h;
  /* aloca nó */
  p = (ListaHet*) malloc(sizeof(ListaHet));
  p->tipo = TRI;
  p->info = t;
  p->prox = NULL;
  return p;
}


/* Cria um nó com um círculo */
ListaHet* cria_cir (float r)
{
  Circulo* c;
  ListaHet* p;
  /* aloca círculo */
  c = (Circulo*) malloc(sizeof(Circulo));
  c->r = r;
  /* aloca nó */
  p = (ListaHet*) malloc(sizeof(ListaHet));
  p->tipo = CIR;
  p->info = c;
  p->prox = NULL;

  return p;
}
```

Uma vez criados os nós, podemos inseri-los na lista, como já vínhamos fazendo com os nós de listas homogêneas.

As constantes simbólicas que representam os tipos dos objetos podem ser agrupadas em uma enumeração (veja o Capítulo 8):

```
enum {
  RET,
  TRI,
  CIR
};
```

Manipulação de listas heterogêneas

Para exemplificar a manipulação de listas heterogêneas, considerando a existência de uma lista com os objetos geométricos apresentados anteriormente, vamos implementar uma função que fornece como valor de retorno a maior área entre os elementos da lista. Uma implementação dessa função é mostrada a seguir. No exemplo, criamos uma função auxiliar que calcula a área do objeto armazenado num determinado nó da lista. Como essa função recebe um nó da lista heterogênea, ela tem de testar o tipo do objeto armazenado. Uma vez identificado o tipo do objeto, a função converte o ponteiro genérico `info` para um ponteiro do tipo específico do objeto em questão. A partir do ponteiro convertido, pode-se acessar os campos da estrutura que define aquele tipo de objeto.

```
#define PI 3.14159

/* função auxiliar: calcula área correspondente ao nó */
static float area (ListaHet* p)
{
  float a;          /* área do elemento */

  switch (p->tipo) {
    case RET:
    {
       /* converte para retângulo e calcula área */
       Retangulo *r = (Retangulo*) p->info;
       a = r->b * r->h;
    }
    break;
```

```
    case TRI:
    {
       /* converte para triângulo e calcula área */
       Triangulo *t = (Triangulo*) p->info;
       a = (t->b * t->h) / 2;
    }
    break;
    case CIR:
    {
       /* converte para círculo e calcula área */
       Circulo *c = (Circulo) p->info;
       a = PI * c->r * c->r;
    }
    break;
  }
  return a;
}
```

Com o auxílio dessa função, podemos escrever o código da função que tem como valor de retorno a maior área dos objetos armazenados na lista:

```
/* Função para cálculo da maior área */
float max_area (ListaHet* l)
{
  float amax = 0.0;     /* maior área */
  ListaHet* p;
  for (p=l; p!=NULL; p=p->prox) {
     float a = area(p);    /* área do nó */
     if (a > amax)
        amax = a;
  }
  return amax;
}
```

Como vemos, a função que acessa a lista não traz nenhuma novidade com relação às funções antes apresentadas para listas homogêneas. Apenas o acesso à informação associada a cada nó é que não pode ser feito de forma direta, pois primeiro precisamos identificar o tipo da informação e então converter o ponteiro genérico para um ponteiro específico.

Para obter um código mais estruturado na manipulação das informações, podemos reescrever a função para o cálculo da área associada a um nó. Vamos agora utilizar mais funções auxiliares, específicas ao cálculo da área de cada objeto geométrico. A função genérica é responsável apenas por chamar a função específica correspondente ao tipo de objeto armazenado no nó:

```
/* função para cálculo da área de um retângulo */
static float ret_area (Retangulo* r)
{
  return r->b * r->h;
}

/* função para cálculo da área de um triângulo */
static float tri_area (Triangulo* t)
{
  return (t->b * t->h) / 2;
}

/* função para cálculo da área de um círculo */
static float cir_area (Circulo* c)
{
  return PI * c->r * c->r;
}

/* função para cálculo da área do nó (versão 2) */
static float area (ListaHet* p)
{
  float a;
  switch (p->tipo) {
     case RET:
        a = ret_area(p->info);
     break;
     case TRI:
        a = tri_area(p->info);
     break;
     case CIR:
        a = cir_area(p->info);
     break;
  }
  return a;
}
```

Nesse caso, a conversão de ponteiro genérico para ponteiro específico é feita quando chamamos uma das funções de cálculo da área: passa-se um ponteiro genérico que é atribuído, por meio de conversão implícita de tipo, a um ponteiro específico[1].

Devemos salientar que, quando trabalhamos com conversão de ponteiros genéricos, temos de garantir que o ponteiro armazene o endereço em que, de fato, existe o tipo específico correspondente. O compilador não tem como verificar se a conversão é válida; a verificação do tipo passa a ser responsabilidade do programador.

[1] Esse código não é válido em C++. A linguagem C++ não tem conversão implícita de um ponteiro genérico para um ponteiro específico. Para compilar em C++, devemos fazer a conversão explicitamente. Por exemplo: `a = ret_area((Retangulo*)p->info);`

11

Pilhas

Uma das estruturas de dados mais simples é a pilha. Possivelmente por essa razão, é a estrutura de dados mais utilizada em programação. Sua idéia fundamental é que todo o acesso a seus elementos seja feito a partir do topo. Assim, quando um elemento novo é introduzido na pilha, ele passa a ser o elemento do topo. O único elemento que pode ser removido da pilha é o do topo.

Para entender o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos. Se quisermos adicionar um prato na pilha, o colocamos no topo. Para pegar um prato da pilha, retiramos o do topo. Assim, temos de retirar o prato do topo para ter acesso ao próximo prato. A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha. Logo, os elementos da pilha só podem ser retirados na ordem inversa à ordem em que foram introduzidos: o primeiro que sai é o último que entrou (a sigla LIFO – *last in, first out* – é usada para descrever essa estratégia).

Existem duas operações básicas que devem ser implementadas em uma estrutura de pilha: a operação para empilhar um novo elemento, inserindo-o no topo, e a operação para desempilhar um elemento, removendo-o do topo. É comum nos referirmos a essas duas operações pelos termos em inglês *push* (empilhar) e *pop* (desempilhar). A Figura 11.1 ilustra o funcionamento conceitual de uma pilha.

O exemplo de utilização de pilha mais próximo é a própria pilha de execução da linguagem C. As variáveis locais das funções são dispostas em uma pilha, e uma função só tem acesso às variáveis da função que está no topo (não é possível acessar as variáveis da função locais às outras funções).

Há várias implementações possíveis de uma pilha, que se distinguem pela natureza dos seus elementos, pela maneira como são armazenados e pelas operações disponíveis para o tratamento da pilha.

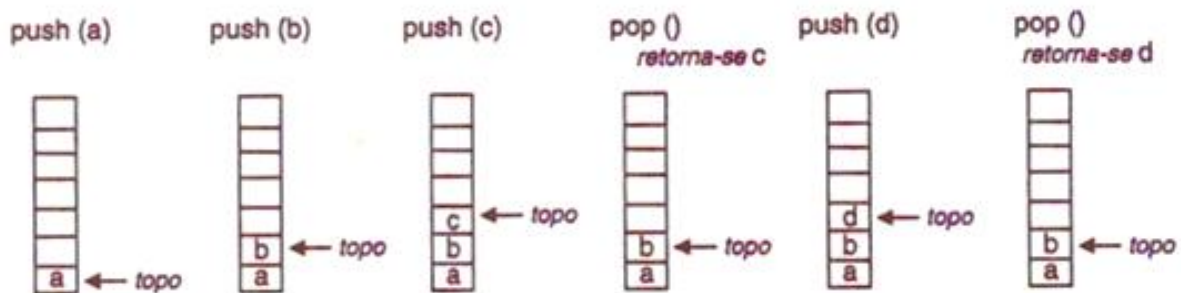



Figura 11.1 *Funcionamento da pilha.*

Interface do tipo pilha

Neste capítulo, consideraremos duas implementações de pilha: usando um vetor e usando uma lista encadeada. Para simplificar a exposição, consideraremos uma pilha que armazena valores reais. De modo independente da estratégia de implementação, podemos definir a interface do tipo abstrato que representa uma estrutura de pilha. Ela é composta pelas operações que estarão disponibilizadas para manipular e acessar as informações da pilha. Neste exemplo, vamos considerar a implementação de cinco operações:

- criar uma pilha vazia;
- inserir um elemento no topo (*push*);
- remover o elemento do topo (*pop*);
- verificar se a pilha está vazia;
- liberar a estrutura de pilha.

O arquivo *pilha.h*, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct pilha Pilha;

Pilha* pilha_cria (void);
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_libera (Pilha* p);
```

A função `cria` aloca dinamicamente a estrutura da pilha, inicializa seus campos e retorna seu ponteiro; as funções `push` e `pop` inserem e retiram, respectivamente, um valor real na pilha; a função `vazia` informa se a pilha está ou não vazia; e a função `libera` destrói a pilha, e assim libera toda a memória usada pela estrutura.

Implementação de pilha com vetor

Em aplicações computacionais que precisam de uma estrutura de pilha, é comum saber de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido. Nesses casos, a implementação da pilha pode ser feita por um vetor, o que é muito simples. Devemos ter um vetor (vet) para armazenar os elementos da pilha, e os elementos inseridos ocupam as primeiras posições do vetor. Dessa forma, se temos n elementos armazenados na pilha, o elemento vet[n-1] representa o do topo.

A estrutura que representa o tipo pilha deve, portanto, ser composta pelo vetor e pelo número de elementos armazenados.

```
#define N  50        /* número máximo de elementos */

struct pilha {
  int n;
  float vet[N];
};
```

A função para criar a pilha aloca dinamicamente essa estrutura e inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

```
Pilha* pilha_cria (void)
{
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->n = 0;   /* inicializa com zero elementos */
  return p;
}
```

Para inserir um elemento na pilha, usamos a próxima posição livre do vetor. Devemos ainda assegurar que existe espaço para a inserção do novo elemento, tendo em vista que se trata de um vetor com dimensão fixa.

```
void pilha_push (Pilha* p, float v)
{
  if (p->n == N) {   /* capacidade esgotada */
     printf("Capacidade da pilha estourou.\n");
     exit(1);          /* aborta programa */
  }
  /* insere elemento na próxima posição livre */
  p->vet[p->n] = v;
  p->n++;
}
```

A função pop retira o elemento do topo da pilha, e fornece seu valor como retorno. Podemos também verificar se a pilha está vazia ou não.

```
float pilha_pop (Pilha* p)
{
  float v;
  if (pilha_vazia(p)) {
     printf("Pilha vazia.\n");
     exit(1);           /* aborta programa */
  }
  /* retira elemento do topo */
  v = p->vet[p->n--1];
  p->n-;
  return v;
}
```

A função que verifica se a pilha está vazia pode ser dada por:

```
int pilha_vazia (Pilha* p)
{
  return (p->n == 0);
}
```

Finalmente, a função para liberar a memória alocada pela pilha pode ser:

```
void pilha_libera (Pilha* p)
{
  free(p);
}
```

Implementação de pilha com lista

Quando o número máximo de elementos que serão armazenados na pilha não é conhecido, devemos implementar a pilha com uma estrutura de dados dinâmica, no caso, com uma lista encadeada. Os elementos são armazenados na lista, e a pilha pode ser representada simplesmente por um ponteiro para o primeiro nó da lista.

O nó da lista para armazenar valores reais pode ser dado por:

```
struct lista {
  float info;
  struct lista* prox;
};
typedef struct lista Lista;
```

A estrutura da pilha é então simplesmente:

```
struct pilha {
  Lista* prim;
};
```

A função cria aloca a estrutura da pilha e inicializa a lista como sendo vazia.

```
Pilha* pilha_cria (void)
{
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->prim = NULL;
  return p;
}
```

O primeiro elemento da lista representa o topo da pilha. Cada novo elemento é inserido no início da lista e, conseqüentemente, sempre que solicitado, retiramos o elemento também do início da lista. A implementação dessas funções é ilustrada a seguir:

```
void pilha_push (Pilha* p, float v)
{
  Lista* n = (Lista*) malloc(sizeof(Lista));
  n->info = v;
  n->prox = p->prim;
  p->prim = n;
}

float pilha_pop (Pilha* p)
{
  Lista* t;
  float v;
  if (pilha_vazia(p)) {
     printf("Pilha vazia.\n");
     exit(1);           /* aborta programa */
  }
  t = p->prim;
  v = t->info;
  p->prim = t->prox;
  free(t);
  return v;
}
```

A pilha estará vazia se a lista estiver vazia:

```
int pilha_vazia (Pilha* p)
{
  return (p->prim==NULL);
}
```

Por fim, a função que libera a pilha deve antes liberar todos os elementos da lista:

```
void pilha_libera (Pilha* p)
{
  Lista* q = p->prim;
  while (q!=NULL) {
     Lista* t = q->prox;
     free(q);
     q = t;
  }
  free(p);
}
```

A rigor, pela definição da estrutura de pilha, só temos acesso ao elemento do topo. No entanto, para testar o código, pode ser útil implementar uma função que imprima os valores armazenados na pilha. Os códigos a seguir ilustram a implementação dessa função nas duas versões de pilha (vetor e lista). A ordem de impressão adotada é do topo para a base.

```
/* imprime: versão com vetor */
void pilha_imprime (Pilha* p)
{
  int i;
  for (i=p->n-1; i>=0; i--)
     printf("%f\n",p->vet[i]);
}

/* imprime: versão com lista */
void pilha_imprime (Pilha* p)
{
  Lista* q;
  for (q=p->prim; q!=NULL; q=q->prox)
     printf("%f\n",q->info);
}
```

Exemplo de uso: calculadora pós-fixada

Um bom exemplo de aplicação de pilha é o funcionamento das calculadoras da HP (Hewlett-Packard). Elas trabalham com expressões pós-fixadas, então para avaliar uma expressão como `(1-2)*(4+5)` podemos digitar 1 2 – 4 5 + *. O funcionamento dessas calculadoras é muito simples. Cada operando é empilhado em uma pilha de valores. Quando se encontra um operador, desempilha-se o número apropriado de operandos (dois para operadores binários e um para operadores unários), realiza-se a operação devida e empilha-se o resultado. Desse modo,

na expressão citada, são empilhados os valores 1 e 2. Quando aparece o operador -, 1 e 2 são desempilhados, e o resultado da operação, no caso -1 (= 1 - 2), é colocado no topo da pilha. A seguir, 4 e 5 são empilhados. O operador seguinte, +, desempilha o 4 e o 5 e empilha o resultado da soma, 9. Nesse momento, estão na pilha os dois resultados parciais, -1 na base e 9 no topo. O operador *, então, desempilha os dois e coloca -9 (= -1 * 9) no topo da pilha.

Como exemplo de aplicação de uma estrutura de pilha, vamos implementar uma calculadora pós-fixada. Ela deve ter uma pilha de valores reais para representar os operandos. Para enriquecer a implementação, vamos considerar o formato com que os valores da pilha são impressos como um dado adicional associado à calculadora. Esse formato pode, por exemplo, ser passado ao criar a calculadora.

Para representar a interface exportada pela calculadora, podemos criar o arquivo *calc.h*:

```
/* Arquivo que define a interface da calculadora */

typedef struct calc Calc;

/* funções exportadas */
Calc* calc_cria (char* f);
void calc_operando (Calc* c, float v);
void calc_operador (Calc* c, char op);
void calc_libera (Calc* c);
```

A implementação da calculadora faz uso do TAD pilha criado anteriormente e independe da implementação usada (vetor ou lista). O tipo que representa a calculadora pode ser dado por:

```
struct calc {
  char f[21];    /* formato para impressão */
  Pilha* p;      /* pilha de operandos */
};
```

A função `cria` recebe como parâmetro de entrada uma cadeia de caracteres com o formato utilizado pela calculadora para imprimir os valores. Essa função cria uma calculadora inicialmente sem operandos na pilha.

```
Calc* calc_cria (char* formato)
{
   Calc* c = (Calc*) malloc(sizeof(Calc));
   strcpy(c->f,formato);
   c->p = pilha_cria( );          /* cria pilha vazia */
   return c;
}
```

A função operando coloca no topo da pilha o valor passado como parâmetro. A função operador retira os dois valores do topo da pilha (só consideraremos operadores binários), efetua a operação correspondente e coloca o resultado no topo da pilha. As operações válidas são: '+' para somar, '-' para subtrair, '*' para multiplicar e '/' para dividir. Se não existirem operandos na pilha, consideraremos que seus valores são zero. Tanto a função operando quanto a função operador imprimem, com a utilização do formato especificado na função cria, o novo valor do topo da pilha.

```
void calc_operando (Calc* c, float v)
{
   /* empilha operando */
   pilha_push(c->p,v);

   /* imprime topo da pilha */
   printf(c->f,v);
}


void calc_operador (Calc* c, char op)
{
   float v1, v2, v;

   /* desempilha operandos */
   if (pilha_vazia(c->p))
      v2 = 0.0;
   else
      v2 = pilha_pop(c->p);
   if (pilha_vazia(c->p))
      v1 = 0.0;
   else
      v1 = pilha_pop(c->p);

   /* faz operação */
   switch (op) {
      case '+':  v = v1+v2;  break;
      case '-':  v = v1-v2;  break;
      case '*':  v = v1*v2;  break;
      case '/':  v = v1/v2;  break;
   }

   /* empilha resultado */
   pilha_push(c->p,v);

   /* imprime topo da pilha */
   printf(c->f,v);
}
```

Por fim, a função para liberar a memória usada pela calculadora libera a pilha de operandos e a estrutura da calculadora.

```
void calc_libera (Calc* c)
{
  pilha_libera(c->p);
  free(c);
}
```

Um programa cliente que faz uso da calculadora é mostrado a seguir:

```
/* Programa para ler expressão e chamar funções da calculadora */

#include <stdio.h>
#include "calc.h"

int main (void)
{
   char c;
   float v;
   Calc* calc;

   /* cria calculadora com formato de duas casas decimais */
   calc = calc_cria ("%.2f\n");

   do {
    /* lê próximo caractere não branco */
    scanf(" %c",&c);
    /* verifica se é operador válido */
    if (c=='+' || c=='-' || c=='*' || c=='/') {
       calc_operador(calc,c);
    }
    /* devolve caractere lido e tenta ler número */
    else {
       ungetc(c,stdin);
       if (scanf("%f",&v) == 1)
          calc_operando(calc,v);
    }
   } while (c!='q');
   calc_libera (calc);
   return 0;
}
```

Esse programa cliente lê os dados fornecidos pelo usuário e opera a calculadora. Para isso, o programa lê um caractere e verifica se é um operador válido. Em caso negativo, o programa "devolve" o caractere lido para o *buffer* de leitura,

usando a função ungetc, e tenta ler um operando. O usuário finaliza a execução do programa digitando q.

Se executado, e considerando-se as expressões digitadas pelo usuário mostradas a seguir, esse programa teria como saída:

```
3 5 8 * +        digitado pelo usuário
3.00
5.00
8.00
40.00
43.00
7 /              digitado pelo usuário
7.00
6.14
q                digitado pelo usuário
```

12

Filas

Outra estrutura de dados bastante usada em computação é a fila. Na estrutura de fila, os acessos aos elementos também seguem uma regra. O que a diferencia da pilha é a ordem de saída dos elementos: enquanto na pilha "o último que entra é o primeiro que sai", na fila "o primeiro que entra é o primeiro que sai" (a sigla FIFO – *first in, first out* – é usada para descrever essa estratégia). A estrutura de fila é uma analogia natural com o conceito de fila que usamos no nosso dia-a-dia: quem primeiro entra numa fila é o primeiro a ser atendido (a sair da fila). Sua idéia fundamental é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início.

Um exemplo de utilização em computação é a implementação de uma fila de impressão. Se uma impressora é compartilhada por várias máquinas, deve-se adotar uma estratégia para determinar que documento será impresso primeiro. A estratégia mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que forem submetidos – o primeiro submetido é o primeiro a ser impresso.

De modo análogo ao que fizemos com a estrutura de pilha, neste capítulo discutiremos duas estratégias para a implementação de uma estrutura de fila: com o uso de um vetor e de uma lista encadeada. Para implementar uma fila, devemos ser capazes de inserir novos elementos em uma extremidade, o fim, e retirar elementos da outra extremidade, o início.

Interface do tipo fila

Antes de discutir as duas estratégias de implementação, podemos definir a interface disponibilizada pela estrutura, isto é, definir quais operações serão implementadas para manipular a fila. Mais uma vez, para simplificar a exposição, consideraremos uma estrutura que armazena valores reais. De maneira independen-

te da estratégia de implementação, a interface do tipo abstrato que representa uma estrutura de fila pode ser composta pelas seguintes operações:

- criar uma fila vazia;
- inserir um elemento no fim;
- retirar o elemento do início;
- verificar se a fila está vazia;
- liberar a fila.

O arquivo *fila.h*, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila Fila;

Fila* fila_cria (void);
void fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int fila_vazia (Fila* f);
void fila_libera (Fila* f);
```

A função `cria` aloca dinamicamente a estrutura da fila, inicializa seus campos e retorna seu ponteiro; a função `insere` adiciona um novo elemento no final da fila, e a função `retira` remove o elemento do início; a função `vazia` informa se a fila está vazia ou não; e a função `libera` destrói a estrutura, e assim libera toda a memória alocada.

Implementação de fila com vetor

Assim como no caso da pilha, nossa primeira implementação de fila será feita usando um vetor para armazenar os elementos. Para isso, devemos fixar o número máximo `N` de elementos na fila. Podemos observar que o processo de inserção e remoção em extremidades opostas fará a fila "andar" no vetor. Por exemplo, se inserirmos os elementos `1.4`, `2.2`, `3.5`, `4.0` e depois retirarmos dois elementos, a fila não estará mais nas posições iniciais do vetor. A Figura 12.1 ilustra a configuração da fila após a inserção dos primeiros quatro elementos, e a Figura 12.2, após a remoção de dois elementos.

Figura 12.1 *Fila após inserção de quatro novos elementos.*

Figura 12.2 *Fila após retirar dois elementos.*

Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada do vetor pode chegar à última posição. Para reaproveitar as primeiras posições livres do vetor sem implementar uma re-arrumação trabalhosa dos elementos, podemos incrementar as posições do vetor de forma "circular": se o último elemento da fila ocupa a última posição do vetor, inserimos os novos elementos a partir do início do vetor. Dessa forma, em um dado momento, poderíamos ter quatro elementos, `20.0`, `20.8`, `21.2` e `24.3`, distribuídos dois no fim do vetor e dois no início.

Figura 12.3 *Fila com incremento circular.*

Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam "circularmente". Desse modo, se temos 100 posições no vetor, os índices assumem os seguintes valores:

```
0, 1, 2, 3, …, 98, 99, 0, 1, 2, 3, …, 98, 99, 0, 1,…
```

Podemos definir uma função auxiliar responsável por incrementar o valor de um índice em uma unidade. Essa função recebe o valor do índice atual e fornece como valor de retorno o índice incrementado, por meio do incremento circular. Uma possível implementação dessa função é:

```
static int incr (int i)
{
  if (i == N-1)
     return 0;
  else
     return i+1;
}
```

Essa mesma função pode ser implementada de uma forma mais compacta, por meio do operador módulo:

```
static int incr(int i)
{
  return (i+1)%N;
}
```

Com o uso do operador módulo, em geral optamos por dispensar a função auxiliar e escrever diretamente o incremento circular:

```
...
i=(i+1)%N;
...
```

Podemos declarar o tipo fila como sendo uma estrutura com três componentes: um vetor `vet` de tamanho `N`, um inteiro `n` que representa o número de elementos armazenados na fila e um índice `ini` para o início da fila.

Conforme ilustrado nas figuras apresentadas, usamos as seguintes convenções para a identificação da fila:

- `ini` marca a posição do próximo elemento a ser retirado da fila;
- `fim` marca a posição (vazia) em que será inserido o próximo elemento.

De posse do índice para o início e do número de elementos, podemos calcular o índice `fim` incrementando `ini` de `n` unidades, também de forma circular:

```
fim = (ini+n)%N.
```

A estrutura de fila pode então ser dada por:

```
#define N 100

struct fila {
  int n;
  int ini;
  float vet[N];
};
```

A função para criar a fila aloca dinamicamente essa estrutura e inicializa a fila como sendo vazia.

```
Fila* fila_cria (void)
{
  Fila* f = (Fila*) malloc(sizeof(Fila));
  f->n = 0;       /* inicializa fila vazia */
  f->ini = 0;     /* escolhe uma posição inicial */
  return f;
}
```

Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por `fim`. Devemos ainda assegurar que há espaço para a inserção do novo elemento, haja vista se tratar de um vetor com capacidade limitada.

```
void fila_insere (Fila* f, float v)
{
  int fim;
  if (f->n == N) {   /* fila cheia: capacidade esgotada */
     printf("Capacidade da fila estourou.\n");
     exit(1);          /* aborta programa */
  }
  /* insere elemento na próxima posição livre */
  fim = (f->ini + f->n) % N;
  f->vet[fim] = v;
  f->n++;
}
```

A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno. Podemos também verificar se a fila está vazia ou não.

```
float fila_retira (Fila* f)
{
  float v;
  if (fila_vazia(f)) {
     printf("Fila vazia.\n");
     exit(1);          /* aborta programa */
  }
  /* retira elemento do início */
  v = f->vet[f->ini];
  f->ini = (f->ini + 1) % N;
  f->n--;
  return v;
}
```

A função que verifica se a fila está vazia pode ser dada por:

```
int fila_vazia (Fila* f)
{
  return (f->n == 0);
}
```

Finalmente, a função para liberar a memória alocada pela fila pode ser:

```
void fila_libera (Fila* f)
{
  free(f);
}
```

Implementação de fila com lista

Vamos agora ver como implementar uma fila usando uma lista encadeada, que será, como nos exemplos anteriores, uma lista simplesmente encadeada, em que cada nó guarda um ponteiro para o próximo nó da lista. Como teremos de inserir e retirar elementos das extremidades opostas da lista, as quais representarão o início e o fim da fila, teremos de usar dois ponteiros, `ini` e `fim`, que apontam respectivamente para o primeiro e para o último elemento da fila. Essa situação é ilustrada na Figura 12.4:

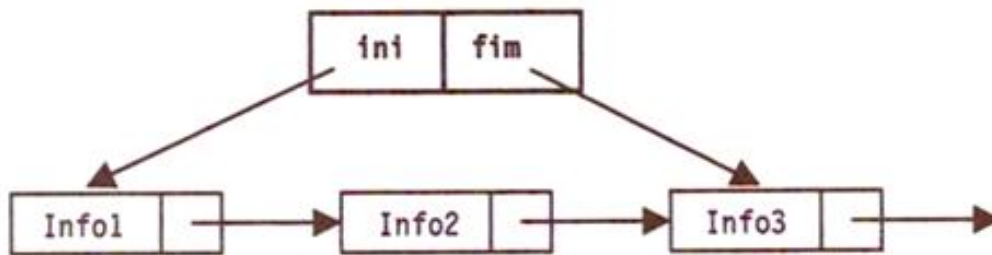



Figura 12.4 *Estrutura de fila com lista encadeada.*

A operação para retirar um elemento ocorre no início da lista (fila) e consiste essencialmente em fazer com que, após a remoção, `ini` aponte para o sucessor do nó retirado. (Observe que seria mais complicado remover um nó do fim da lista simplesmente encadeada, porque o antecessor não é encontrado com a mesma facilidade que seu sucessor.) A inserção também é simples, pois basta acrescentar à lista um sucessor para o último nó, apontado por `fim`, e fazer com que `fim` aponte para esse novo nó.

O nó da lista para armazenar valores reais, como já vimos, pode ser dado por:

```
struct lista {
  float info;
  struct lista* prox;
};
typedef struct lista Lista;
```

A estrutura da fila agrupa os ponteiros para o início e o fim da lista:

```
struct fila {
  Lista* ini;
  Lista* fim;
};
```

A função `cria` aloca a estrutura da fila e inicializa a lista como sendo vazia.

```
Fila* fila_cria (void)
{
  Fila* f = (Fila*) malloc(sizeof(Fila));
  f->ini = f->fim = NULL;
  return f;
}
```

Cada novo elemento é inserido no fim e, sempre que solicitado, retiramos o elemento do início da lista. Dessa forma, precisamos de dois procedimentos: para inserir no fim e para remover do início. O procedimento para inserir no fim ainda não foi discutido, mas é simples, uma vez que temos explicitamente armazenado o ponteiro para o último elemento. Devemos salientar que a função de inserção deve atualizar os dois ponteiros, `ini` e `fim`, no momento da inserção do primeiro elemento. Analogamente, a função para retirar deve atualizá-los se a fila tornar-se vazia após a remoção do elemento:

```
void fila_insere (Fila* f, float v)
{
  Lista* n = (Lista*) malloc(sizeof(Lista));
  n->info = v;          /* armazena informação */
  n->prox = NULL;       /* novo nó passa a ser o último */
  if (f->fim != NULL)   /* verifica se lista não estava vazia */
     f->fim->prox = n;
  else                  /* fila estava vazia */
     f->ini = n;
  f->fim = n;           /* fila aponta para novo elemento */
}

float fila_retira (Fila* f)
{
  Lista* t;
  float v;
  if (fila_vazia(f)) {
     printf("Fila vazia.\n");
     exit(1);          /* aborta programa */
  }
  t = f->ini;
  v = t->info;
  f->ini = t->prox;
  if (f->ini == NULL)   /* verifica se fila ficou vazia */
     f->fim = NULL;
  free(t);
  return v;
}
```

A fila estará vazia se a lista estiver vazia:

```
int fila_vazia (Fila* f)
{
  return (f->ini==NULL);
}
```

Por fim, a função que libera a fila deve antes liberar todos os elementos da lista.

```
void fila_libera (Fila* f)
{
  Lista* q = f->ini;
  while (q!=NULL) {
     Lista* t = q->prox;
     free(q);
     q = t;
  }
  free(f);
}
```

Analogamente à pilha, para testar o código, pode ser útil implementar uma função que imprima os valores armazenados na fila. Os códigos a seguir ilustram a implementação dessa função nas duas versões de fila (vetor e lista). A ordem de impressão adotada é do início para o fim.

```
/* imprime: versão com vetor */
void fila_imprime (Fila* f)
{
  int i;
  for (i=0; i<f->n; i++)
     printf("%f\n",f->vet[(f->ini+i)%N]);
}

/* imprime: versão com lista */
void fila_imprime (Fila* f)
{
  Lista* q;
  for (q=f->ini; q!=NULL; q=q->prox)
     printf("%f\n",q->info);
}
```

Um exemplo simples de utilização da estrutura de fila é apresentado a seguir:

```
/* Módulo para ilustrar utilização da fila */

#include <stdio.h>
#include "fila.h"

int main (void)
{
  Fila* f = fila_cria( );
```

```
    fila_insere(f,20.0);
    fila_insere(f,20.8);
    fila_insere(f,21.2);
    fila_insere(f,24.3);
    printf("Primeiro elemento: %f\n", fila_retira(f));
    printf("Segundo elemento: %f\n", fila_retira(f));
    printf("Configuracao da fila:\n");
    fila_imprime(f);
    fila_libera(f);
    return 0;
}
```

Fila dupla

A estrutura de dados que chamamos de fila dupla consiste em uma fila na qual é possível inserir novos elementos nas duas extremidades, no início e no fim. Conseqüentemente, permite-se também retirar elementos dos dois extremos. É como se, dentro de uma mesma estrutura de fila, tivéssemos duas filas, com os elementos dispostos em ordem inversa uma da outra.

A interface do tipo abstrato que representa uma fila dupla acrescenta novas funções para inserir e retirar elementos. Podemos enumerar as seguintes operações:

- criar uma estrutura de fila dupla;
- inserir um elemento no início;
- inserir um elemento no fim;
- retirar o elemento do início;
- retirar o elemento do fim;
- verificar se a fila está vazia;
- liberar a fila.

O arquivo *fila2.h*, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila2 Fila2;

Fila2* fila2_cria (void);
void fila2_insere_ini (Fila2* f, float v);
void fila2_insere_fim (Fila2* f, float v);
float fila2_retira_ini (Fila2* f);
float fila2_retira_fim (Fila2* f);
int fila2_vazia (Fila2* f);
void fila2_libera (Fila2* f);
```

A implementação dessa estrutura por meio de um vetor para armazenar os elementos não traz grandes dificuldades, pois o vetor permite acesso randômico aos elementos. Vamos analisar as duas novas funções: `insere_ini` e `retira_fim`. Para inserir no início, devemos inserir o elemento no índice que precede `ini`, adotando um decremento circular. O índice precedente pode ser obtido assim:

```
static int decr (int i)
{
  i--;
  if (i<0)
     return N-1;
  else
     return i;
}
```

Esse decremento circular também pode ser feito de forma mais compacta:

```
static int decr (int i)
{
  return (i-1+N)%N;
}
```

Dessa maneira, a função para inserir no início pode ser dada por:

```
void fila2_insere_ini (Fila* f, float v)
{
  int prec;
  if (f->n == N) {   /* fila cheia: capacidade esgotada */
     printf("Capacidade da fila estourou.\n");
     exit(1);          /* aborta programa */
  }
  /* insere elemento na posição precedente ao início */
  prec = (f->ini - 1 + N) % N;
  f->vet[prec] = v;
  f->ini = prec;       /* atualiza índice para início */
  f->n++;
}
```

Para retirar do final da fila, devemos acessar o último elemento armazenado na fila. De posse de `ini` e `n`, o índice do último elemento é dado por: `(ini+n-1)%N`. Assim, uma possível implementação da função que retira do final pode ser:

```
float fila2_retira_fila (Fila* f)
{
  int ult;
  float v;
```

```
    if (fila2_vazia(f)) {
       printf("Fila vazia.\n");
       exit(1);          /* aborta programa */
    }
    /* retira último elemento */
    ult = (f->ini + f->n - 1) % N;
    v = f->vet[ult];
    f->n--;
    return v;
}
```

Implementação de fila dupla com lista

A implementação de uma fila dupla com lista encadeada merece uma discussão mais detalhada. A dificuldade que encontramos reside na implementação da função para retirar um elemento do final da lista. Todas as outras funções já foram discutidas e poderiam ser implementadas sem dificuldade com o uso de uma lista simplesmente encadeada. No entanto, na lista encadeada, a função para retirar do fim não pode ser implementada de forma eficiente, pois, dado o ponteiro para o último elemento da lista, não temos como acessar o anterior, que passaria a ser o último elemento.

Para solucionar esse problema, temos de lançar mão da estrutura de lista duplamente encadeada (veja o Capítulo 10). Nessa lista, cada nó guarda, além da referência para o próximo elemento, uma referência para o elemento anterior: dado o ponteiro de um nó, podemos acessar os elementos adjacentes. Esse arranjo resolve o problema de acessar o elemento anterior ao último. Devemos salientar que o uso de uma lista duplamente encadeada para implementar a fila é simples, pois só manipulamos os elementos das extremidades da lista.

O arranjo de memória para implementar a fila dupla com lista é ilustrado na Figura 12.5:

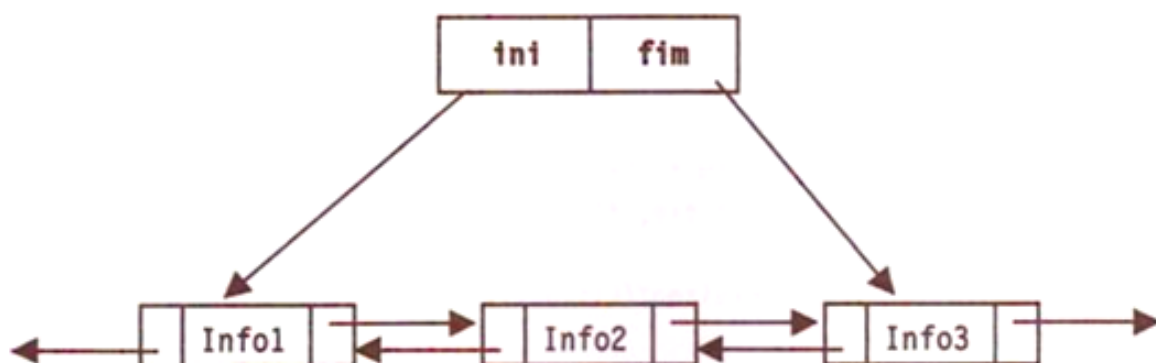



Figura 12.5 *Arranjo da estrutura de fila dupla com lista.*

O nó da lista duplamente encadeada para armazenar valores reais pode ser dado por:

```
struct lista2 {
  float info;
  struct lista2* ant;
  struct lista2* prox;
};
typedef struct lista2 Lista2;
```

A estrutura da fila dupla agrupa os ponteiros para o início e o fim da lista:

```
struct fila2 {
  Lista2* ini;
  Lista2* fim;
};
```

Interessa-nos discutir as funções para inserir e retirar elementos. As demais são praticamente idênticas às de fila simples. Podemos inserir um novo elemento em qualquer extremidade da fila. Aqui, vamos optar por definir duas funções auxiliares de lista: para inserir no início e para inserir no fim. Ambas as funções são simples e já foram exaustivamente discutidas para o caso da lista simples. No caso da lista duplamente encadeada, a diferença consiste em termos de atualizar também o encadeamento para o elemento anterior. Uma possível implementação dessas funções é mostrada a seguir. Essas funções retornam, respectivamente, o novo nó inicial e final.

```
/* função auxiliar: insere no início */
static Lista2* ins2_ini (Lista2* ini, float v)
{
  Lista* p = (Lista2*) malloc(sizeof(Lista2));
  p->info = v;
  p->prox = ini;
  p->ant = NULL;
  if (ini != NULL)   /* verifica se lista não estava vazia */
     ini->ant = p;
  return p;
}

/* função auxiliar: insere no fim */
static Lista2* ins2_fim (Lista2* fim, float v)
{
  Lista2* p = (Lista2*) malloc(sizeof(Lista2));
  p->info = v;
  p->prox = NULL;
  p->ant = fim;
  if (fim != NULL)   /* verifica se lista não estava vazia */
     fim->prox = p;
  return p;
}
```

Uma possível implementação das funções auxiliares para remover o elemento do início ou do fim é mostrada a seguir. Essas funções também retornam, respectivamente, o novo nó inicial e final.

```
/* função auxiliar: retira do início */
static Lista2* ret2_ini (Lista2* ini)
{
  Lista2* p = ini->prox;
  if (p != NULL)   /* verifica se lista não ficou vazia */
     p->ant = NULL;
  free(ini);
  return p;
}

/* função auxiliar: retira do fim */
static Lista2* ret2_fim (Lista2* fim)
{
  Lista2* p = fim->ant;
  if (p != NULL)   /* verifica se lista não ficou vazia */
     p->prox = NULL;
  free(fim);
  return p;
}
```

As funções que manipulam a fila fazem uso dessas funções de lista, e atualizam os ponteiros `ini` e `fim` quando necessário.

```
void fila2_insere_ini (Fila2* f, float v)
{
  f->ini = ins2_ini(f->ini,v);
  if (f->fim==NULL)  /* fila antes vazia? */
     f->fim = f->ini;
}

void fila2_insere_fim (Fila2* f, float v)
{
  f->fim = ins2_fim(f->fim,v);
  if (f->ini==NULL)  /* fila antes vazia? */
     f->ini = f->fim;
}

float fila2_retira_ini (Fila2* f)
{
  float v;
  if (fila2_vazia(f)) {
     printf("Fila vazia.\n");
     exit(1);            /* aborta programa */
  }
```

```
  v = f->ini->info;
  f->ini = ret2_ini(f->ini);
  if (f->ini == NULL)   /* fila ficou vazia? */
     f->fim = NULL;
  return v;
}

float fila2_retira_fim (Fila2* f)
{
  float v;
  if (fila_vazia(f)) {
     printf("Fila vazia.\n");
     exit(1);         /* aborta programa */
  }
  v = f->fim->info;
  f->fim = ret2_fim(f->fim);
  if (f->fim == NULL)   /* fila ficou vazia? */
     f->ini = NULL;
  return v;
}
```

Por fim, lembramos que a implementação de tipos abstratos pode ser feita com a utilização de tipos abstratos já existentes. Nesse sentido, se temos os tipos abstratos que representam listas, poderíamos construir os tipos abstratos de pilha e fila com os tipos de lista. Fica como exercício reescrever os tipos de pilha e fila com os TADs de listas.

13

Árvores

Nos capítulos anteriores, examinamos as estruturas de dados que podem ser chamadas de lineares, como vetores e listas. A importância dessas estruturas é inegável, mas elas não são adequadas para representar dados que devem ser dispostos de maneira hierárquica. Por exemplo, os arquivos (documentos) que criamos em um computador são armazenados dentro de uma estrutura hierárquica de diretórios (pastas). Existe um diretório base dentro do qual podemos armazenar diversos subdiretórios e arquivos. Por sua vez, dentro deles, podemos armazenar outros subdiretórios e arquivos, e assim por diante, recursivamente.

Neste capítulo, apresentaremos as árvores, estruturas de dados adequadas para a representação de hierarquias. A forma mais natural de definir uma estrutura de árvore é usando a recursividade. Uma árvore é composta por um conjunto de nós. Existe um nó r, denominado raiz, que contém zero ou mais subárvores, cujas raízes são ligadas diretamente a r. Esses nós raízes das subárvores são ditos filhos do nó pai, r. Nós com filhos são comumente chamados de nós internos, e nós que não têm filhos são chamados de folhas ou nós externos. É tradicional desenhar as estruturas de árvores com a raiz para cima e as folhas para baixo. A Figura 13.1 exemplifica a estrutura de uma árvore.

Figura 13.1 *Estrutura de árvore.*

Observamos que, por adotar essa forma de representação gráfica, não representamos explicitamente a direção dos ponteiros, subentendendo que eles apontam sempre do pai para os filhos.

O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os vários tipos de árvores existentes. Neste capítulo, estudaremos dois tipos. Primeiro, examinaremos as árvores binárias, nas quais cada nó tem, no máximo, dois filhos. Depois examinaremos as estruturas de árvores nas quais o número de filhos é variável. Estruturas recursivas serão usadas como base para o estudo e a implementação das operações com árvores.

Árvores binárias

Um exemplo de utilização de árvores binárias é a avaliação de expressões. Como trabalhamos com operadores que esperam um ou dois operandos, os nós da árvore para representar uma expressão têm no máximo dois filhos. Nessa árvore, os nós folhas representam operandos, e os nós internos, operadores. Uma árvore que representa, por exemplo, a expressão (3+6)*(4-1)+5 é ilustrada na Figura 13.2.

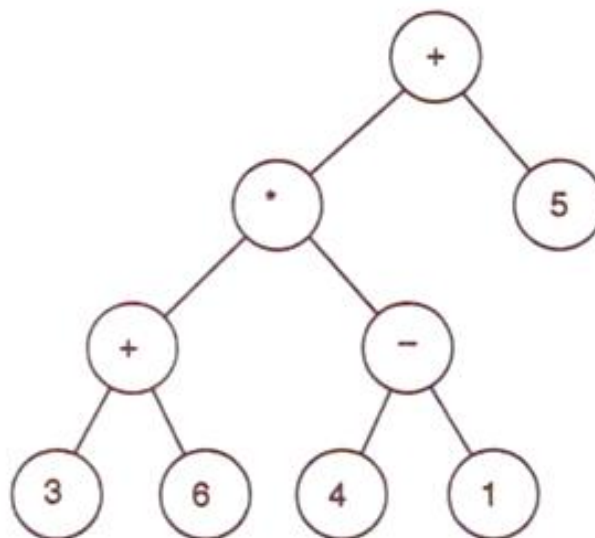



Figura 13.2 *Árvore da expressão: (3+6) * (4-1) + 5.*

Em uma árvore binária, cada nó tem zero, um ou dois filhos. De maneira recursiva, podemos definir uma árvore binária como sendo:

- uma árvore vazia; ou
- um nó raiz tendo duas subárvores, identificadas como a subárvore da direita (*sad*) e a subárvore da esquerda (*sae*).

A Figura 13.3 ilustra a definição de árvore binária. Essa definição recursiva será usada na construção de algoritmos e na verificação (informal) da correção e do seu desempenho.

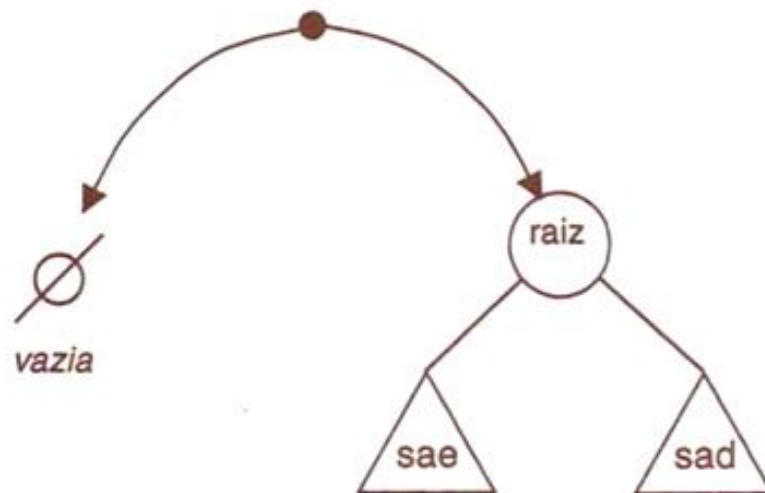



Figura 13.3 *Representação esquemática da definição da estrutura de árvore binária.*

A Figura 13.4, a seguir, ilustra um exemplo de árvore binária. Os nós *a*, *b*, *c*, *d*, *e*, *f* formam uma árvore binária da seguinte maneira: a árvore é composta pelo nó *a*, pela subárvore à esquerda formada por *b* e *d*, e pela subárvore à direita formada por *c*, *e* e *f*. O nó *a* representa a raiz da árvore, e os nós *b* e *c*, as raízes das subárvores. Finalmente, os nós *d*, *e* e *f* são folhas da árvore. Devemos notar que cada nó folha também representa uma árvore, com duas subárvores vazias.

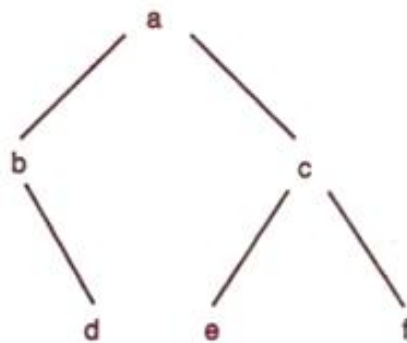



Figura 13.4 *Exemplo de árvore binária.*

Para descrever árvores binárias, podemos usar a seguinte notação textual: a árvore vazia é representada por < >, e árvores não-vazias, por `<raiz sae sad>`. Com essa notação, a árvore da Figura 13.4 é representada por:

```
<a<b< ><d< >< >>><c<e< >< >><f< >< >>>>
```

Pela definição, uma subárvore de uma árvore binária é sempre especificada como sendo a *sae* ou a *sad* de uma árvore maior, e qualquer das duas subárvores pode ser vazia. Assim, as duas árvores da Figura 13.5 são distintas.

Isso também pode ser visto pelas representações textuais das duas árvores, que são, respectivamente: `<a<b< >< >>< >>` e `<a< ><b< >< >>>`.

Figura 13.5 *Duas árvores binárias distintas.*

Representação em C

De modo análogo ao que fizemos para as demais estruturas de dados, podemos definir um tipo para representar uma árvore binária. Para simplificar a discussão, vamos considerar as informações que queremos armazenar nos nós da árvore como sendo de valores de caracteres simples. Vamos inicialmente discutir como podemos representar uma estrutura de árvore binária em C. Que estrutura podemos usar para representar um nó da árvore? Cada nó deve armazenar três informações: a informação propriamente dita, no caso um caractere, e dois ponteiros para as subárvores, à esquerda e à direita. Então a estrutura de C para representar o nó da árvore pode ser dada por:

```
struct arv {
  char info;
  struct arv* esq;
  struct arv* dir;
};
```

Da mesma forma que uma lista encadeada é representada por um ponteiro para o primeiro nó, a estrutura da árvore é representada por um ponteiro para o nó raiz. Dado o ponteiro para o nó raiz da árvore, tem-se acesso aos demais nós.

Como acontece com qualquer TAD (tipo abstrato de dados), as operações que fazem sentido para uma árvore binária dependem essencialmente da forma de utilização da árvore. Nesta seção, em vez de discutir a interface do tipo abstrato para depois mostrar sua implementação, vamos optar por discutir algumas operações com a exibição simultânea de suas implementações. Ao final da seção, apresentaremos um arquivo que pode representar a interface do tipo. Nas funções seguintes, consideraremos que existe o tipo `Arv`, definido por:

```
typedef struct arv Arv;
```

Como veremos, as funções que manipulam árvores são, em geral, implementadas de forma recursiva, por meio da definição recursiva da estrutura.

Vamos procurar identificar e descrever apenas operações cuja utilidade seja a mais geral possível. Uma operação que provavelmente deverá ser incluída em todos os casos é a de criação de uma árvore vazia. Como uma árvore é representada pelo endereço do nó raiz, uma árvore vazia tem de ser representada pelo valor `NULL`. Assim, a função que cria uma árvore vazia pode ser simplesmente:

```
Arv* arv_criavazia (void)
{
  return NULL;
}
```

Para construir árvores não-vazias, podemos ter uma operação que cria um nó raiz dadas a informação e as duas subárvores, a da esquerda e a da direita. Essa função tem como valor de retorno o endereço do nó raiz criado e pode ser dada por:

```
Arv* arv_cria (char c, Arv* sae, Arv* sad)
{
  Arv* p=(Arv*)malloc(sizeof(Arv));
  p->info = c;
  p->esq = sae;
  p->dir = sad;
  return p;
}
```

As duas funções para a criação de árvores, `criavazia` e `cria`, representam os dois casos da definição recursiva de árvore binária: uma árvore binária (`Arv* a;`) é vazia (`a=arv_criavazia( );`) ou é composta por uma raiz e duas subárvores (`a=arv_cria(c,sae,sad);`). Assim, de posse dessas duas funções, podemos criar árvores mais complexas.

Para exemplificar, podemos verificar que a árvore ilustrada na Figura 13.4 pode ser criada pela seguinte seqüência de atribuições:

```
/* sub-árvore 'd' */
Arv* a1= arv_cria('d',arv_criavazia( ),arv_criavazia( ));
/* sub-árvore 'b' */
Arv* a2= arv_cria('b',arv_criavazia( ),a1);
/* sub-árvore 'e' */
Arv* a3= arv_cria('e',arv_criavazia( ),arv_criavazia( ));
/* sub-árvore 'f' */
Arv* a4= arv_cria('f',arv_criavazia( ),arv_criavazia( ));
/* sub-árvore 'c' */
Arv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
Arv* a = arv_cria('a',a2,a5 );
```

Como alternativa, a árvore poderia ser criada com uma única atribuição, seguindo a sua estrutura, "recursivamente":

```
Arv* a = arv_cria('a',
            arv_cria('b',
                arv_criavazia( ),
                arv_cria('d', arv_criavazia( ), arv_criavazia( ))
            ),
            arv_cria('c',
                arv_cria('e', arv_criavazia( ), arv_criavazia( )),
                arv_cria('f', arv_criavazia( ), arv_criavazia( ))
            )
           );
```

Para tratar a árvore vazia de forma diferente das outras, é importante ter uma operação que diz se uma árvore é ou não vazia. Podemos ter:

```
int arv_vazia (Arv* a)
{
  return a==NULL;
}
```

Outra função muito útil consiste em exibir o conteúdo da árvore. Essa função deve percorrer recursivamente a árvore, visitando todos os nós e imprimindo sua informação. A implementação dessa função usa a definição recursiva da árvore. Vimos que uma árvore binária ou é vazia ou é composta pela raiz e por duas subárvores. Portanto, para imprimir a informação de todos os nós da árvore, devemos primeiro testar se ela é vazia. Se não for, imprimimos a informação associada à raiz e chamamos (recursivamente) a função para imprimir as subárvores.

```
void arv_imprime (Arv* a)
{
  if (!arv_vazia(a)){
     printf("%c ", a->info);          /* mostra raiz */
     arv_imprime(a->esq);             /* mostra sae */
     arv_imprime(a->dir);             /* mostra sad */
  }
}
```

Assim, se a função `imprime` fosse aplicada à arvore ilustrada na Figura 13.4, a saída da função seria: `a b d c e f`.

Podemos modificar a implementação de `imprime` de forma que a saída impressa reflita, além do conteúdo de cada nó, a estrutura da árvore, por meio da notação textual apresentada anteriormente. Uma possível implementação dessa função é mostrada a seguir:

```
void arv_imprime (Arv* a)
{
  printf("<");
```

```
  if (!arv_vazia(a)){
     printf("%c", a->info);          /* mostra raiz */
     arv_imprime(a->esq);            /* mostra sae */
     arv_imprime(a->dir);            /* mostra sad */
  }
  printf(">");
}
```

Outra operação que pode ser acrescentada é a operação para liberar a memória alocada pela estrutura da árvore. Mais uma vez, usaremos uma implementação recursiva. Um cuidado especial a ser tomado é liberar as subárvores antes de liberar o nó raiz, para que o acesso a elas não seja perdido antes de serem removidas. Nesse caso, vamos optar por fazer com que a função tenha como valor de retorno a árvore atualizada, isto é, uma árvore vazia, representada por `NULL`.

```
Arv* arv_libera (Arv* a){
  if (!arv_vazia(a)){
     arv_libera(a->esq);   /* libera sae */
     arv_libera(a->dir);   /* libera sad */
     free(a);              /* libera raiz */
  }
  return NULL;
}
```

Devemos notar que a definição de árvore, por ser recursiva, não faz distinção entre árvores e subárvores. Assim, `cria` pode ser usada para acrescentar ("enxertar") uma subárvore em um ramo de uma árvore, e `libera` pode ser usada para remover ("podar") uma subárvore qualquer de uma árvore dada.

Dessa forma, se considerarmos a criação da árvore feita anteriormente:

```
Arv* a = arv_cria('a',
            arv_cria('b',
                 arv_criavazia( ),
                 arv_cria('d', arv_criavazia( ), arv_criavazia( ))
                ),
            arv_cria('c',
                 arv_cria('e', arv_criavazia( ), arv_criavazia( )),
                 arv_cria('f', arv_criavazia( ), arv_criavazia( ))
                )
          );
```

podemos acrescentar alguns nós, com:

```
a->esq->esq = arv_cria('x',
                 arv_cria('y',arv_criavazia( ),arv_criavazia( )),
                 arv_cria('z',arv_criavazia( ),arv_criavazia( ))
                );
```

e podemos liberar alguns outros, com:

```
a->dir->esq = arv_libera(a->dir->esq);
```

Deixamos como exercício a verificação do resultado final dessas operações. No entanto, é importante observar que, de modo análogo ao que fizemos para retirar um elemento de uma lista, o código cliente que chama a função `libera` é responsável por atribuir o valor atualizado retornado pela função, no caso uma árvore vazia. No exemplo anterior, se não tivéssemos feito a atribuição, o endereço armazenado em `r->dir->esq` seria o de uma área de memória não mais em uso.

Outra função que podemos considerar percorre a árvore para verificar a ocorrência de um determinado caractere c em um de seus nós. Essa função tem como retorno um valor booleano (um ou zero) que indica a ocorrência ou não do caractere na árvore.

```
int arv_pertence (Arv* a, char c){
  if (arv_vazia(a))
     return 0;    /* árvore vazia: não encontrou */
  else
     return a->info==c ||
            arv_pertence(a->esq,c) ||
            arv_pertence(a->dir,c);
}
```

Note que essa forma de programar `pertence` em C, usando o operador lógico `||` ("ou"), interrompe a busca tão logo o elemento seja encontrado. Isso acontece porque se `c==a->info` for verdadeiro, as duas outras expressões não chegam a ser avaliadas. Analogamente, se o caractere for encontrado na subárvore da esquerda, a busca não prossegue na subárvore da direita.

Podemos dizer que a expressão:

```
return c==a->info ||
      arv_pertence(a->esq,c) ||
      arv_pertence(a->dir,c);
```

é equivalente a:

```
if (c==a->info)
  return 1;
else if (arv_pertence(a->esq,c))
  return 1;
else
  return arv_pertence(a->dir,c);
```

Finalmente, considerando que as funções discutidas e implementadas formam a interface do tipo abstrato para representar uma árvore binária, um arquivo de interface *arv.h* pode ser dado por:

```
typedef struct arv Arv;

Arv* arv_criavazia (void);
Arv* arv_cria (char c, Arv* e, Arv* d);
Arv* arv_libera (Arv* a);
int  arv_vazia (Arv* a);
int  arv_pertence (Arv* a, char c);
void arv_imprime (Arv* a);
```

Ordens de percurso em árvores binárias

A programação da operação `imprime` vista anteriormente seguiu a ordem empregada na definição de árvore binária para decidir a ordem em que as três ações seriam executadas: imprimimos o conteúdo da raiz, em seguida imprimimos o conteúdo da subárvore à esquerda e, então, imprimimos o conteúdo da subárvore à direita. Entretanto, dependendo da aplicação em vista, essa ordem poderia não ser a preferível, podendo ser utilizada uma outra ordem, por exemplo:

```
arv_imprime(a->esq);             /* mostra sae */
arv_imprime(a->dir);             /* mostra sad */
printf("%c ", a->info);          /* mostra raiz */
```

Muitas operações em árvores binárias envolvem o percurso de todas as subárvores, com a execução de alguma ação de tratamento em cada nó, de forma que é comum percorrer uma árvore em uma das seguintes ordens:

- *pré-ordem*: trata *raiz*, percorre *sae*, percorre *sad*;
- *ordem simétrica*: percorre *sae*, trata *raiz*, percorre *sad*;
- *pós-ordem*: percorre *sae*, percorre *sad*, trata *raiz*.

Na implementação da função `libera`, por exemplo, tivemos de adotar a pós-ordem:

```
arv_libera(a->esq);  /* libera sae */
arv_libera(a->dir);  /* libera sad */
free(a);             /* libera raiz */
```

Na terceira parte deste livro, quando tratarmos de árvores binárias de busca, apresentaremos um exemplo de aplicação de árvores binárias em que a ordem de percurso importante é a ordem simétrica. Algumas outras ordens de percurso po-

dem ser definidas, mas a maioria das aplicações envolve uma dessas três ordens, percorrendo a *sae* antes da *sad*.

Como exercício, sugerimos implementar diferentes versões da função `imprime`, percorrendo a árvore em ordem simétrica e em pós-ordem. Pode-se verificar o resultado da aplicação dessas duas funções na árvore da Figura 13.4.

Altura de uma árvore

Uma propriedade fundamental de todas as árvores é que só existe um caminho da raiz para qualquer nó. Com isso, podemos definir a altura de uma árvore como sendo o comprimento do caminho mais longo da raiz até uma das folhas. Por exemplo, a altura da árvore da Figura 13.4 é 2 e a altura das árvores da Figura 13.5 é 1. Assim, a altura de uma árvore com um único nó raiz é zero e, por conseguinte, dizemos que a altura de uma árvore vazia é negativa e vale -1. Também podemos numerar os níveis em que os nós aparecem na árvore. A raiz está no nível 0, seus filhos diretos no nível 1, e assim por diante. O último nível da árvore é o nível *h*, sendo *h* a altura da árvore.

Uma árvore binária é dita cheia (ou completa) se todos os seus nós internos têm duas subárvores associadas e todos os nós folhas estão no último nível. A Figura 13.6 ilustra uma árvore cheia. Podemos notar que nesse tipo de árvore temos um nó no nível 0, dois nós no nível 1, quatro nós no nível 2, oito nós no nível 3, e assim por diante. Isto é, no nível n, temos 2^n nós. Também podemos notar que o número de nós de um determinado nível de uma árvore cheia é uma unidade a mais do que a soma de todos os nós dos níveis anteriores:

$$2^n = 1 + \sum_{i=0}^{n-1} 2^i$$

É possível então mostrar que uma árvore cheia de altura h tem um número de nós dado por: $2^{h+1} - 1$.

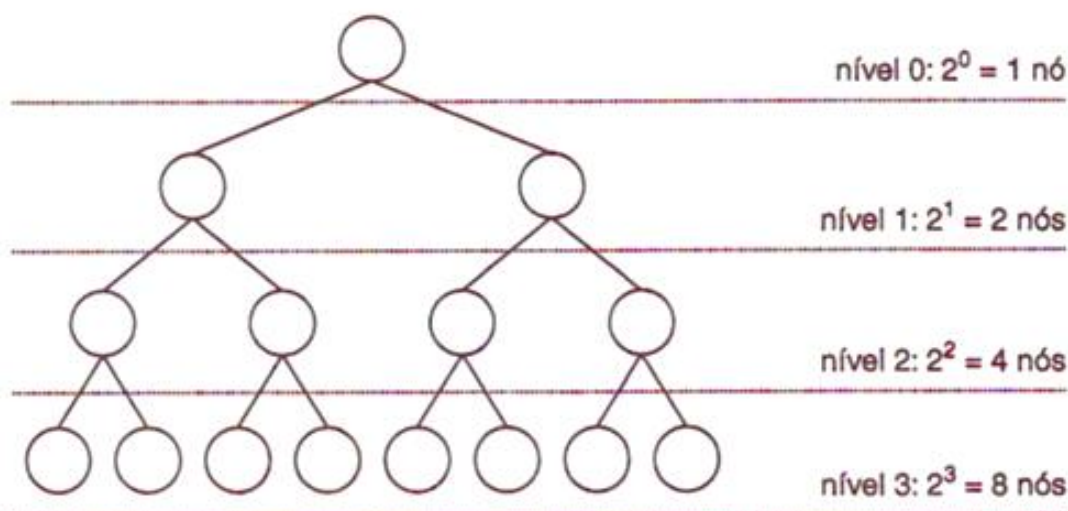



Figura 13.6 *Árvore binária cheia.*

Uma árvore é dita degenerada se todos os seus nós internos têm uma única subárvore associada. De fato, a estrutura hierárquica se degenera em uma estrutura linear. A Figura 13.7 exemplifica árvores degeneradas. Observamos que, em uma árvore degenerada, temos um único nó em cada nível. Assim, uma árvore degenerada de altura h tem $h+1$ nós.

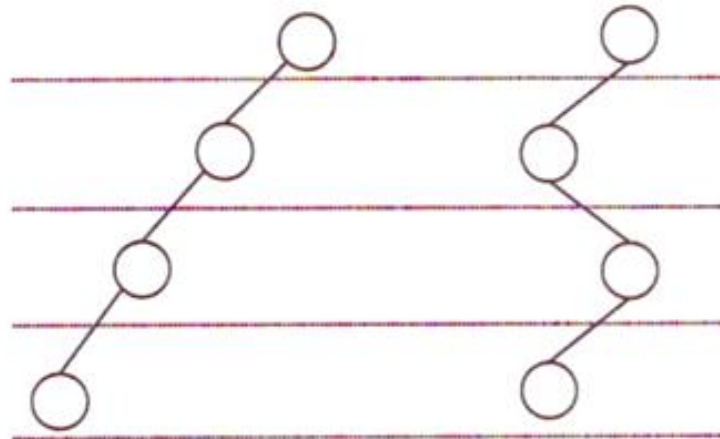



Figura 13.7 *Árvores binárias degeneradas.*

A altura de uma árvore é uma medida importante na avaliação da eficiência com que visitamos os nós de uma árvore. Uma árvore binária com n nós tem uma altura mínima proporcional a $log\,n$ (caso da árvore cheia) e uma altura máxima proporcional a n (caso da árvore degenerada). A altura indica o esforço computacional necessário para alcançar qualquer nó da árvore. Quando discutirmos árvores binárias de busca, verificaremos a importância de manter as árvores com altura pequena, isto é, manter as árvores com uma distribuição dos nós próxima à da árvore cheia.

Podemos pensar na implementação de uma função que calcula a altura de uma árvore binária. A implementação dessa função é simples: basta aplicar a definição recursiva dada. Se a árvore for vazia, sua altura, por definição, vale -1. Se a árvore não for vazia, sua altura será dada pela maior altura das subárvores acrescida de 1 (a árvore tem um nível a mais do que suas subárvores, que é o nível da sua raiz). Uma possível implementação dessa função, que usa uma função auxiliar para calcular o máximo entre dois números inteiros, é mostrada a seguir:

```
static int max2 (int a, int b)
{
  return (a > b) ? a : b;
}

int arv_altura (Arv* a)
{
  if (arv_vazia(a))
     return -1;
  else
     return 1 + max2(arv_altura(a->esq),arv_altura(a->dir));
}
```

Devemos notar que a altura da árvore vazia deve ser -1 para a função acima funcionar corretamente.

Árvores com número variável de filhos

Nesta seção, discutiremos as estruturas de árvores com número variável de filhos. Como vimos, numa árvore binária, o número de filhos dos nós é limitado em, no máximo, dois. Vamos agora considerar as estruturas de árvores nas quais cada nó pode ter mais do que duas subárvores associadas. Como as subárvores de um determinado nó formam um conjunto linear e são dispostas em uma determinada ordem, faz sentido falar em primeira subárvore (sa_1), segunda subárvore (sa_2) etc. A Figura 13.8 ilustra um exemplo de árvore no qual o número máximo de filhos não está limitado a dois.

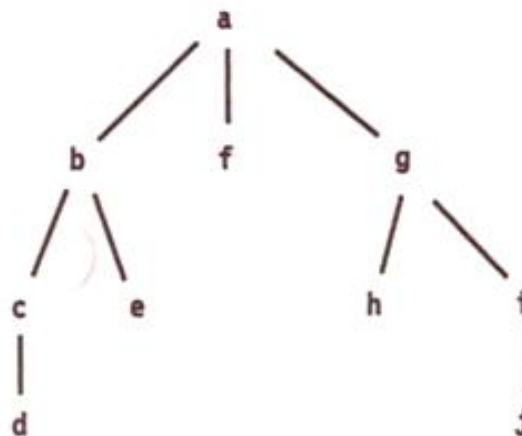



Figura 13.8 *Exemplo de árvore que não é binária.*

Nesse exemplo, podemos notar que a árvore com raiz no nó `a` tem 3 subárvores, ou seja, o nó `a` tem 3 filhos. Os nós `b` e `g` têm dois filhos cada um; os nós `c` e `i` têm um filho cada, e os nós `d`, `e`, `h` e `j` são folhas e têm zero filhos.

De forma semelhante ao que foi feito no caso das árvores binárias, podemos representar essas árvores com notação textual, usando o seguinte formato:

```
<raiz sa1 sa2 ... san>
```

Com essa notação, a árvore da Figura 13.8 seria representada por:

```
<a <b <c <d>> <e>> <f> <g <h> <i <j>>>>
```

Representação em C

Dependendo da aplicação, podemos usar várias estruturas para representar árvores, levando em consideração o número de filhos que cada nó pode apresentar.

Se soubermos, por exemplo, que em uma aplicação o número máximo de filhos a ser apresentado por um nó é 3, podemos montar uma estrutura com 3 campos de apontadores para os nós filhos, digamos, `f1`, `f2` e `f3`. Os campos não utilizados podem ser preenchidos com o valor nulo `NULL`. Ao prever um número máximo de filhos igual a 3 e considerar a implementação de árvores para armazenar valores de caracteres simples, a declaração do tipo que representa o nó da árvore poderia ser:

```
struct arv3 {
  char info;
  struct arv3 *f1, *f2, *f3;
};
```

A Figura 13.9 ilustra a representação da árvore da Figura 13.8 com essa organização.

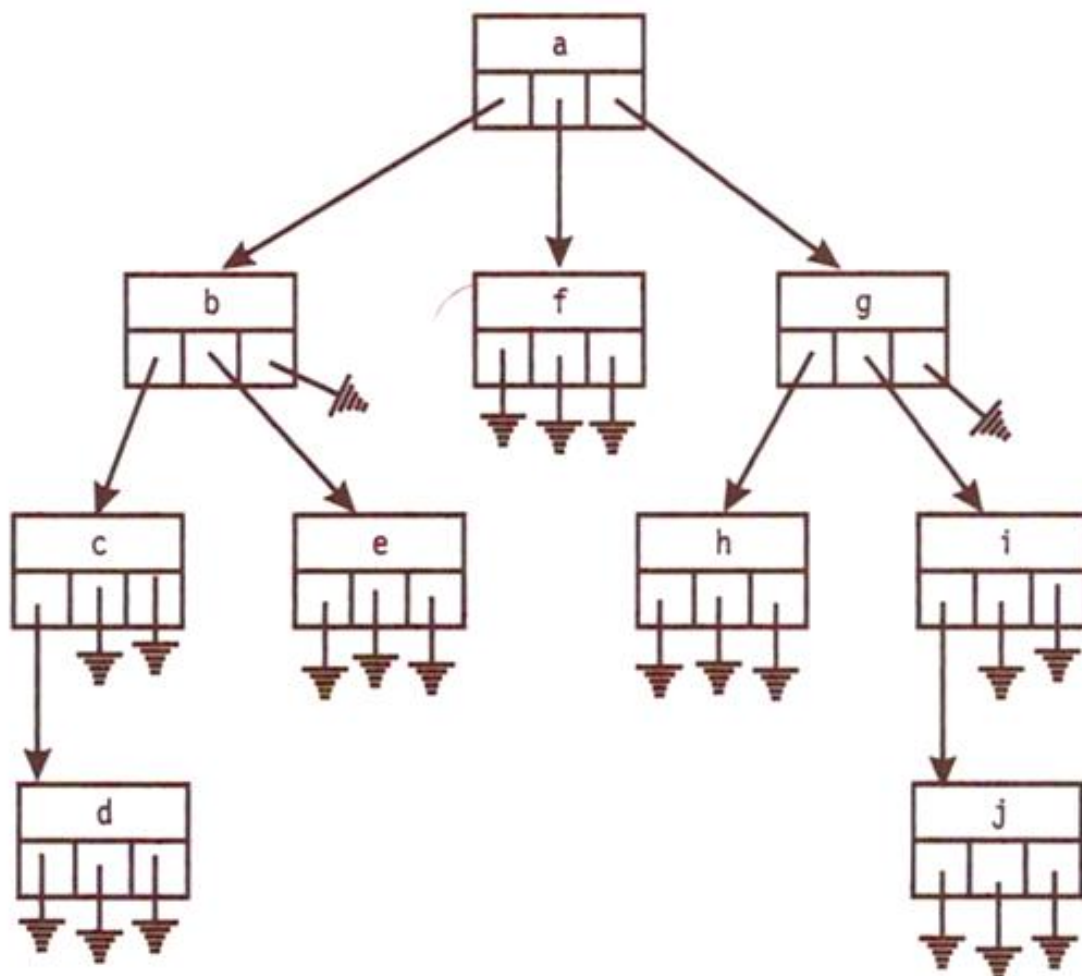



Figura 13.9 *Árvore com no máximo três filhos por nó.*

Para ilustrar o acesso aos elementos da árvore com essa representação em C, podemos implementar uma função para exibir a representação textual no formato mostrado anteriormente (considerando `Arv3` sinônimo para `struct arv3`):

```
void arv3_imprime (Arv3* a)
{
  if (a != NULL)
  {
     printf("<%c",a->info);
     arv3_imprime(a->f1);
     arv3_imprime(a->f2);
     arv3_imprime(a->f3);
     printf(">");
  }
}
```

Apesar de correto, o código dessa função mostra que a representação não é muito adequada, pois não existe uma maneira sistemática de acessar os nós filhos. Existem diversas aplicações computacionais em que precisamos trabalhar com árvores nas quais o número de filhos é limitado. Na área de Computação Gráfica, por exemplo, são muito utilizadas as árvores com quatro e com oito filhos por nó, conhecidas como *quadtree* e *octree*. Portanto, precisamos estruturar de maneira mais adequada os filhos dos nós da árvore (seria impraticável declarar um campo na estrutura para cada possível filho).

Uma representação mais adequada consiste em armazenar os filhos dos nós em um vetor. Assim, a representação do nó para a árvore com até três filhos passa a ser dada por:

```
#define N 3

struct arv3 {
  char info;
  struct no *f[N];
};
```

Com essa representação, temos uma maneira sistemática de visitar todos os filhos de um nó. A nova função para exibir o conteúdo da árvore pode ser dada por:

```
void arv3_imprime (Arv3* a)
{
  if (a != NULL)
  {
     int i;
     printf("<%c",a->info);
     for (i=0; i<N; i++)
        arv3_imprime(a->f[i]);
     printf(">");
  }
}
```

Com isso, o mesmo código pode ser aplicado a árvores com outros limites de número de filhos; basta alterar o valor da constante simbólica `N`.

No entanto, em aplicações em que não existe um limite superior no número de filhos, essa técnica não é aplicável. O mesmo acontece se existe um limite no número de nós, mas esse limite é raramente alcançado, pois estaríamos tendo um grande desperdício de espaço de memória com os campos não utilizados. Nesses casos, precisamos, de fato, de uma estrutura de árvore que não imponha restrições ao número de filhos de cada nó. Um exemplo de aplicação é a representação de árvores de diretórios, na qual o número de filhos varia arbitrariamente.

A representação em C de uma árvore com número variável de filhos por nó pode utilizar então uma "lista de filhos": um nó aponta apenas para seu primeiro (`prim`) filho, e cada um de seus filhos aponta para o próximo (`prox`) irmão. Dessa forma, cada nó pode ter um número arbitrário de filhos. A Figura 13.10 ilustra essa representação de árvore.

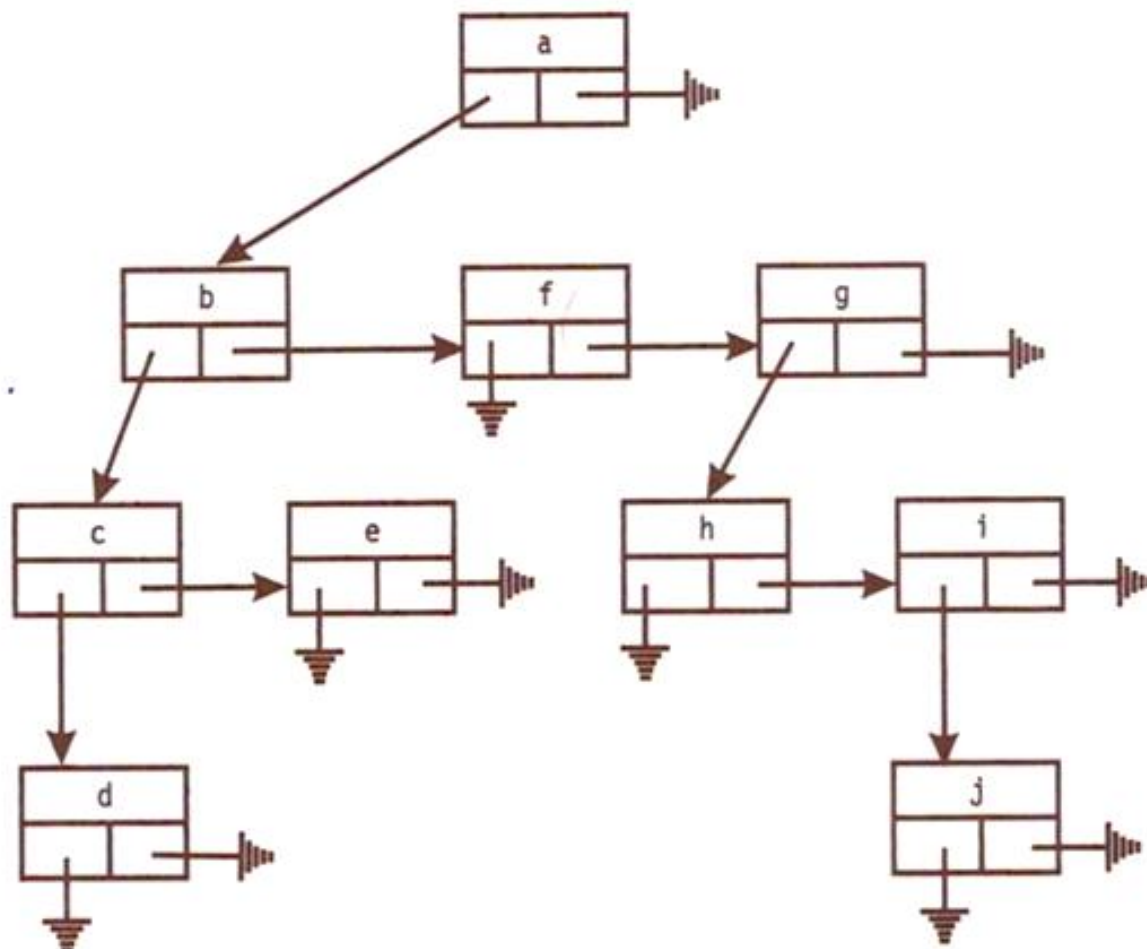



Figura 13.10 *Representação de árvores com "lista de filhos".*

Devemos notar que essa representação também permite acessar de modo sistemático os filhos de um nó, pois eles estão organizados em uma estrutura de lista encadeada.

A composição do tipo que representa um nó dessa árvore pode ser dada por:

```
struct arvvar {
  char info;
  struct arvvar *prim; /* ponteiro para eventual primeiro filho */
  struct arvvar *prox; /* ponteiro para eventual irmão */
};

typedef struct arvvar ArvVar;
```

Portanto, cada nó, além da informação associada, guarda duas referências: uma para a primeira subárvore filha e outra para a próxima subárvore irmã. Se o nó representar uma folha da árvore, o valor de `prim` será `NULL`, pois esse nó não terá filhos. Se o nó representar o último filho de outro nó, o valor de `prox` será `NULL`, pois não existirá um próximo irmão.

As funções que manipulam esse tipo de árvore serão implementadas de forma recursiva. Na implementação dessas funções, adotaremos a seguinte definição de árvore.

Uma árvore é composta por:

- um nó raiz; e
- zero ou mais subárvores.

A Figura 13.11 ilustra o esquema dessa definição.

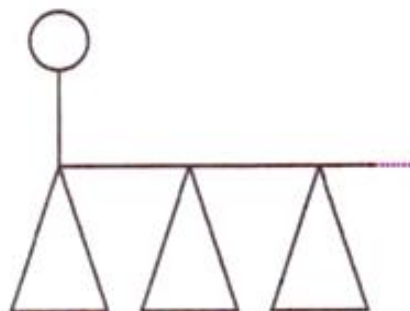



Figura 13.11 *Representação gráfica de uma árvore com número variável de filhos.*

Estritamente, segundo essa definição, uma árvore não pode ser vazia, e a árvore vazia não é sequer mencionada na definição. Assim, uma folha de uma árvore não é um nó com subárvores vazias, como no caso da árvore binária, mas é um nó com zero subárvores. Em qualquer definição recursiva, deve haver uma "condição de contorno", que permita a definição de estruturas finitas, e, no nosso caso, a definição de uma árvore se encerra nas folhas, identificadas como nós com zero subárvores.

Como as funções implementadas nesta seção vão se basear nessa definição, não será considerado o caso de árvores vazias. Essa pequena restrição simplifica as implementações recursivas e, em geral, não limita a utilização da estrutura em aplicações reais. Uma árvore de diretório, por exemplo, nunca é vazia, pois sempre existe o diretório base – o diretório raiz.

Tipo abstrato de dados

Para exemplificar a implementação de funções que manipulam uma árvore com número variável de filhos, vamos considerar a criação de um tipo abstrato de dados para representar árvores em que a informação associada a cada nó é um caractere simples. Podemos definir o seguinte conjunto de operações:

- cria um nó folha, dada a informação a ser armazenada;
- insere uma nova subárvore como filha de um dado nó;
- percorre todos os nós e imprime suas informações;
- verifica a ocorrência de um determinado valor em um dos nós da árvore;
- libera toda a memória alocada pela árvore.

A interface do tipo pode então ser definida no arquivo *arvvar.h*, dado por:

```
typedef struct arvvar ArvVar;

ArvVar* arvv_cria (char c);
void    arvv_insere (ArvVar* a, ArvVar* sa);
void    arvv_imprime (ArvVar* a);
int     arvv_pertence (ArvVar* a, char c);
void    arvv_libera (ArvVar* a);
```

Vamos então apresentar a implementação de cada uma dessas funções. A estrutura `arvvar`, que representa o nó da árvore, é definida conforme mostrado anteriormente. A função para criar uma folha deve alocar o nó e inicializar seus campos, com a atribuição de `NULL` aos campos `prim` e `prox`, pois se trata de um nó folha isolado.

```
ArvVar* arvv_cria (char c)
{
  ArvVar *a =(ArvVar *) malloc(sizeof(ArvVar));
  a->info = c;
  a->prim = NULL;
  a->prox = NULL;
  return a;
}
```

A função que insere uma nova subárvore como filha de um dado nó é muito simples. Como não vamos atribuir nenhum significado especial à posição de um nó filho, a operação de inserção pode inserir a subárvore em qualquer posição. Nesse caso, vamos optar por inserir sempre no início da lista que, como já vimos, é a maneira mais simples de inserir um novo elemento em uma lista encadeada.

```
void arvv_insere (ArvVar* a, ArvVar* sa)
{
  sa->prox = a->prim;
  a->prim = sa;
}
```

Com essas duas funções, podemos construir a árvore do exemplo da Figura 13.10 com o seguinte fragmento de código:

```
/* cria nós como folhas */
ArvVar* a = arvv_cria('a');
ArvVar* b = arvv_cria('b');
ArvVar* c = arvv_cria('c');
ArvVar* d = arvv_cria('d');
ArvVar* e = arvv_cria('e');
ArvVar* f = arvv_cria('f');
ArvVar* g = arvv_cria('g');
ArvVar* h = arvv_cria('h');
ArvVar* i = arvv_cria('i');
ArvVar* j = arvv_cria('j');
/* monta a hierarquia */
arvv_insere(c,d);
arvv_insere(b,e);
arvv_insere(b,c);
arvv_insere(i,j);
arvv_insere(g,i);
arvv_insere(g,h);
arvv_insere(a,g);
arvv_insere(a,f);
arvv_insere(a,b);
```

Para imprimir as informações associadas aos nós da árvore, temos duas opções para percorrê-la: pré-ordem, primeiro a raiz e depois as subárvores, ou pós-ordem, primeiro as subárvores e depois a raiz. Note que, nesse caso, não faz sentido a ordem simétrica, pois o número de subárvores é variável. Para essa função, vamos optar por imprimir o conteúdo dos nós em pré-ordem:

```
void arvv_imprime (ArvVar* a)
{
  ArvVar* p;
  printf("<%c\n",a->info);
```

```
  for (p=a->prim; p!=NULL; p=p->prox)
     arvv_imprime(p);   /* imprime cada sub-árvore filha */
  printf(">");
}
```

A operação para verificar a ocorrência de uma dada informação na árvore é exemplificada a seguir:

```
int arvv_pertence (ArvVar* a, char c)
{
  ArvVar* p;
  if (a->info==c)
     return 1;
  else {
     for (p=a->prim; p!=NULL; p=p->prox) {
         if (arvv_pertence(p,c))
             return 1;
     }
     return 0;
  }
}
```

A última operação apresentada é a que libera a memória alocada pela árvore. O único cuidado que precisamos tomar na programação dessa função é liberar as subárvores antes de liberar o espaço associado a um nó (isto é, usar pós-ordem).

```
void arvv_libera (ArvVar* a)
{
  ArvVar* p = a->prim;
  while (p!=NULL) {
     ArvVar* t = p->prox;
     arvv_libera(p);
     p = t;
  }
  free(a);
}
```

Altura da árvore

As mesmas definições de níveis e altura que fizemos para as árvores binárias se aplicam às árvores com número variável de filhos. Assim, a árvore ilustrada na Figura 13.10 tem altura igual a 3, pois os nós `d` e `j` estão no nível 3 da árvore. A seguir, mostraremos a implementação de uma função que calcula a altura de uma árvore desse tipo.

Para o cálculo da altura da árvore, devemos considerar a definição recursiva usada em nossas implementações. Dessa forma, a altura da árvore será uma unidade a mais do que a maior altura entre as subárvores filhas. Portanto, precisamos computar a maior altura entre elas e retornar esse valor acrescido de uma unidade. Caso o nó raiz não tenha filhos, a altura da árvore deve ser zero. Uma implementação dessa função pode ser dada por:

```
int arvv_altura (ArvVar* a)
{
  int hmax = -1;    /* -1 para tratar caso c/ zero filhos */
  ArvVar* p;

  for (p=a->prim; p!=NULL; p=p->prox) {
     int h = arvv_altura(p);
     if (h > hmax)
         hmax = h;
  }
  return hmax + 1;
}
```

Topologia binária

A representação de árvores com número variável de filhos que fizemos é apenas conceitual. Concretamente, a estrutura usada para representar o nó da árvore adota a mesma topologia que usamos para representar o nó da árvore binária. O nó, além da informação associada, tem dois ponteiros para subárvores. O que muda é o significado atribuído a cada uma das subárvores referenciadas. No caso da árvore binária, uma representava a subárvore à esquerda, e outra, à direita. Aqui, uma representa a primeira subárvore filha, e outra, a subárvore irmã. A Figura 13.12 ilustra a topologia binária de ambas as representações.

Figura 13.12 *Topologia binária dos nós de árvores: binária e com número variável de filhos.*

Feita essa observação, podemos trabalhar com a definição de árvore com número variável de filhos de maneira análoga à que fizemos para árvore binária. Podemos tratar uma árvore como sendo:

- uma árvore vazia; ou
- um nó raiz com duas subárvores, identificadas como a subárvore filha e a subárvore irmã.

Com essa nova definição de árvore, podemos reescrever os códigos das funções discutidas. Algumas funções ficam mais simples se feitas usando essa nova definição. O leitor deve optar por utilizar a definição que julga mais adequada para resolver o problema em questão. Para ilustrar, vamos escrever a função que calcula a altura de uma árvore usando essa nova definição recursiva. Assim, a altura da árvore será o maior valor entre a altura da subárvore filha acrescido de uma unidade e a altura da subárvore irmã. O código dessa implementação é mostrado a seguir. Devemos notar que o caso da árvore vazia agora deve ser considerado, pois faz parte da definição recursiva que estamos usando.

```
static max2 (int a, int b)
{
  return (a > b) ? a : b;
}

int arvv_altura (ArvVar* a)
{
  if (a==NULL)
     return -1;
  else
     return max2(1+arvv_altura(a->prim), arvv_altura(a->prox));
}
```

Funções implementadas para árvores binárias que não diferenciam as árvores referenciadas podem ser diretamente aplicadas a árvores com número variável de filhos. Por exemplo, se tivermos implementado uma função para calcular o número de nós presentes em uma árvore binária, essa mesma função servirá para calcular o número de nós de uma árvore com número variável de filhos.

14

Estruturas genéricas

Nos capítulos anteriores, apresentamos as estruturas de dados básicas utilizadas para a organização de informações na memória do computador. Na apresentação das estruturas e das principais funções de acesso, consideramos que a informação associada a cada nó era representada por um tipo simples. Dessa forma, foi possível concentrar a discussão na estrutura em si, já que o tratamento do dado associado era trivial. Todas as estruturas vistas e todas as funções discutidas podem ser aplicadas aos casos em que precisamos armazenar informações mais complexas. De fato, já foram apresentadas técnicas de programação que podem ser utilizadas para a representação de informações estruturadas em vetores e listas (veja os Capítulos 8 e 10).

Essas mesmas técnicas podem ser aplicadas às outras estruturas de dados. No entanto, para cada novo tipo que quiséssemos tratar, teríamos de reimplementar as funções responsáveis por manipular a estrutura. Como já discutimos, as funções mantêm-se praticamente idênticas; é necessário apenas modificar o tratamento das informações associadas a cada nó. Podemos então pensar em construir estruturas de dados genéricas, isto é, estruturas de dados capazes de armazenar qualquer tipo de informação. Para tanto, o tipo abstrato de dados (TAD) deve desconhecer a natureza da informação associada e ser responsável apenas por sua manutenção e seu encadeamento na estrutura.

O cliente de um TAD de tipo genérico fica responsável por todas as operações que envolvam o acesso direto às informações. Internamente, o TAD guarda apenas um ponteiro para a informação. Esse ponteiro deve ser do tipo genérico, pois não se sabe, a princípio, o tipo da informação que será armazenada. Como já vimos no Capítulo 10, quando discutimos a implementação de listas heterogêneas, um ponteiro genérico em C é representado pelo tipo `void*`. Assim, o TAD, de posse de um ponteiro genérico, não pode acessar a memória por ele apontada, já que não conhece a informação armazenada. Por sua vez, o cliente pode converter

esse ponteiro genérico no ponteiro específico para o tipo em questão e, então, acessar os dados do tipo.

Lista genérica

Vamos exemplificar a implementação de um TAD de tipo genérico por meio de uma lista simplesmente encadeada. As mesmas técnicas de programação podem ser aplicadas às demais estruturas de dados.

A estrutura do nó de uma lista genérica tem de guardar o ponteiro para a informação e o ponteiro para o próximo nó. O código a seguir ilustra essa representação:

```
struct listagen {
  void* info;
  struct listagen* prox;
};

typedef struct listagen ListaGen;
```

Funções que não manipulam a informação associada aos nós podem ser implementadas da forma já vista. Por exemplo, a função para criar uma lista vazia e a função para testar se uma lista está vazia são idênticas às funções já apresentadas para listas de valores inteiros.

Funções que manipulam a informação como um objeto opaco, isto é, funções que não precisam acessar as informações, não oferecem dificuldades na implementação. Por exemplo, podemos implementar uma função que insere um novo nó no início da lista. O cliente é responsável por passar para a função o ponteiro da informação que será armazenada nesse novo nó. Portanto, a função para inserção não precisa acessar a informação, basta fazer o novo nó ter como informação o ponteiro passado pelo cliente. A implementação dessa função é similar ao caso da lista de inteiros, o que varia é apenas o tipo do parâmetro passado (agora um ponteiro genérico).

```
ListaGen* lgen_insere (ListaGen* l, void* p)
{
  ListaGen* n = (ListaGen*) malloc(sizeof(ListaGen));
  n->info = p;
  n->prox = l;
  return n;
}
```

A dificuldade aparece quando temos de implementar as funções que precisam ter acesso às informações dos nós. Por exemplo, como podemos implementar uma função que libera a estrutura? Se quisermos liberar a estrutura da lista,

provavelmente também vamos querer liberar as informações associadas aos nós. No entanto, como o TAD desconhece as informações, o cliente é quem deve ser responsável por liberá-las. O TAD deve ficar responsável apenas por liberar a estrutura de dados em si.

Uma situação similar é observada para a implementação da função que verifica se uma determinada informação está presente na lista. O TAD não é capaz de testar a igualdade entre duas informações. Apenas comparar os ponteiros não resolve, pois devemos comparar as informações propriamente ditas. O cliente pode, por exemplo, associar aos nós informações dos alunos de uma disciplina e pode desejar fazer uma busca baseada apenas no nome de um aluno.

Se considerássemos a implementação de uma função para imprimir as informações da lista, teríamos o mesmo problema: apenas o cliente é capaz de acessar as informações e exibi-las na tela. Na verdade, como não sabemos *a priori* o tipo de informação que será associado aos nós, não podemos nem prever quais funções deverão ser oferecidas na interface do TAD. Cada cliente associará um tipo de informação diferente e precisará de funções específicas para processar as informações armazenadas na estrutura. Por exemplo, um cliente que armazena pontos geométricos pode precisar de uma função para calcular o centro geométrico dos pontos armazenados. Um outro cliente que armazena dados relativos aos alunos de uma disciplina pode precisar de uma função para calcular a média obtida numa prova.

Portanto, o TAD deve prover uma função genérica para percorrer todos os nós da estrutura. Para cada nó visitado, devemos implementar um mecanismo que permita chamar o cliente passando a informação associada. O cliente então processa a informação com finalidades específicas para cada situação.

Uso de *callbacks*

Nosso objetivo é implementar uma função que percorra todos os elementos armazenados na estrutura genérica. Para tanto, devemos separar a função que percorre os elementos da ação que será realizada a cada elemento. Assim, a função que percorre os elementos é única e pode ser usada para diversos fins. A ação a ser executada é passada como parâmetro e é geralmente chamada de *callback*, pois é uma função do cliente (quem usa a função que percorre os elementos) que é "chamada de volta" a cada elemento encontrado na estrutura de dados. Em geral, essa função *callback* recebe como parâmetro a informação associada ao elemento encontrado na estrutura. No nosso exemplo, como temos uma lista genérica, a função receberia o ponteiro para cada informação encontrada na lista.

Para definir como parâmetro qual função *callback* deve ser chamada, temos de apresentar o conceito de ponteiro para função. O nome de uma função representa o endereço dessa função. A nossa função *callback* teria a seguinte assinatura:

```
void callback (void* info);
```

Uma variável ponteiro para armazenar o endereço dessa função é declarada como:

```
void (*cb) (void*);
```

em que `cb` representa uma variável do tipo ponteiro para funções com a mesma assinatura da função `callback` acima.

Assim, uma função genérica para percorrer os elementos da lista do nosso exemplo pode ser dada por:

```
void lgen_percorre (ListaGen* l, void (*cb)(void*))
{
  ListaGen* p;
  for (p=l; p!=NULL; p=p->prox) {
     cb(p->info);
  }
}
```

Isto é, para cada elemento visitado, chama-se a função do cliente passando como parâmetro a informação associada.

Um exemplo de cliente

Para ilustrar a utilização da lista genérica, vamos considerar uma aplicação cliente que armazena pontos (x,y) na estrutura. O tipo `Ponto` pode ser definido por:

```
struct ponto {
  float x, y;
};
typedef struct ponto Ponto;
```

Para inserir pontos na lista genérica, o cliente aloca dinamicamente uma estrutura do tipo `Ponto` e passa seu ponteiro para a função de inserção. Para encapsular esse procedimento, o cliente pode implementar uma função auxiliar a fim de inserir pontos (x,y) na estrutura da lista genérica.

```
static ListaGen* insere_ponto (ListaGen* l, float x, float y)
{
  Ponto* p = (Ponto*) malloc(sizeof(Ponto));
  p->x = x;
  p->y = y;
  return lgen_insere(l,p);
}
```

Uma vez construída a lista genérica, podemos pensar em imprimir os pontos armazenados na estrutura. Para tanto, vamos fazer uso da função `percorre` discutida anteriormente. A cada elemento visitado, vamos imprimir as coordenadas do ponto associado. Nossa função *callback* é então responsável por converter o ponteiro genérico em um ponteiro para `Ponto` e imprimir a informação. Uma possível implementação dessa *callback* é mostrada a seguir. Devemos notar que seu protótipo é fixo e independe da informação. A conversão para o tipo específico ocorre dentro da função.

```
static void imprime (void* info)
{
  Ponto* p = (Ponto*)info;
  printf("%f  %f\n", p->x, p->y);
}
```

Com isso, para imprimir os elementos da lista bastaria chamar a função `percorre` com a ação acima passada como parâmetro.:

```
  ...
  lgen_percorre(l,imprime);
  ...
```

Essa mesma função `percorre` pode ser usada para, por exemplo, calcular o centro geométrico dos pontos armazenados na lista. A ação associada aqui precisa apenas contar o número de elementos visitados e acumular os valores das coordenadas dos pontos visitados. Para isso, podemos usar variáveis globais com o objetivo de representar o número de elementos e o somatório das coordenadas. A cada chamada da *callback*, esses valores devem ser atualizados. Como `NP` e `CG` são variáveis globais do tipo `int` e `Ponto`, respectivamente, a ação para acumular a soma das coordenadas dos elementos da lista pode ser simplesmente:

```
static void centro_geom (void* info)
{
  Ponto* p = (Ponto*)info;
  CG.x += p->x;
  CG.y += p->y;
  NP++;
}
```

De posse dessa *callback*, o cliente pode calcular o centro geométrico dos pontos:

```
  ...
  NP = 0;
  CG.x = CG.y = 0.0f;
```

```
  lgen_percorre(l,centro_geom);
  CG.x /= NP;
  CG.y /= NP;
  ...
```

Passando dados para a callback

Já mencionamos que o uso de variáveis globais deve, sempre que possível, ser evitado, pois esse uso indiscriminado torna um programa ilegível e difícil de ser mantido. Para evitar o uso de variáveis globais nesses casos, devemos criar um mecanismo para transferir um dado do cliente para a função *callback*. A função que percorre os elementos não manipula esse dado, apenas o transfere para a função *callback*. Como não sabemos *a priori* o tipo de dado que será necessário, nós definimos a função recebendo dois parâmetros: a informação do elemento visitado e um ponteiro genérico com um dado qualquer. O cliente chama a função que percorre os elementos passando como parâmetros a função *callback* e o ponteiro a ser repassado para essa mesma *callback* a cada elemento visitado.

Vamos exemplificar o uso dessa estratégia reimplementando a função que percorre os elementos.

```
void lgen_percorre(ListaGen* l, void(*cb)(void*,void*), void* dado)
{
  ListaGen* p;
  for (p=l; p!=NULL; p=p->prox) {
     cb(p->info,dado);
  }
}
```

Devemos notar que a assinatura da função *callback* foi alterada, pois agora ela recebe dois parâmetros. Podemos usar essa nova versão da função percorre para calcular o centro geométrico dos pontos, sem usar variáveis globais. Primeiro temos de criar um tipo que agrupa os dados necessários para calcular o centro geométrico: o número de pontos e as coordenadas acumuladas:

```
struct cg
{
  int n;
  Ponto p;
};
typedef struct cg Cg;
```

Podemos então redefinir a *callback*, a qual, nesse caso, receberá um ponteiro para um tipo `Cg` que representa a estrutura acima.

```
static void centro_geom (void* info, void* dado)
{
  Ponto* p = (Ponto*)info;
  Cg* cg = (Cg*)dado;
  cg->p.x += p->x;
  cg->p.y += p->y;
  cg->n++;
}
```

Dessa forma, a chamada por parte do cliente pode ser exemplificada por este trecho de código:

```
  ...
  Cg cg = {0,{0.0f,0.0f}};
  lgen_percorre(l,centro_geom,&cg);
  cg.p.x /= cg.n;
  cg.p.y /= cg.n;
  ...
```

Retornando valores de callbacks

Vamos agora considerar que queremos verificar a ocorrência de um determinado ponto de coordenadas (x,y) na estrutura da lista. Nesse caso, nossa função *callback* pode receber como dado adicional as coordenadas do ponto que queremos encontrar. No entanto, da forma que a função `percorre` está implementada, todos os elementos da lista serão visitados, mesmo se encontrarmos o ponto de interesse entre os primeiros elementos da lista.

Para evitar esse esforço computacional desnecessário, devemos criar um mecanismo para permitir ao cliente interromper a visitação aos elementos. Esse mecanismo pode ser implementado fazendo com que a função *callback* tenha um valor de retorno. Podemos, por exemplo, adotar a seguinte convenção: se a *callback* tiver zero como valor de retorno, a função deve prosseguir e visitar o próximo elemento; se ela tiver um valor diferente de zero como retorno, a função `percorre` deve interromper a visitação aos elementos e ter esse valor fornecido pela *callback* como seu retorno. Portanto, a assinatura da função `percorre` também muda, pois passa a ter um valor de retorno: zero se não houve interrupção e diferente de zero se houve interrupção por parte do cliente. Uma possível implementação dessa nova versão da função `percorre` é mostrada a seguir:

```
int lgen_percorre (ListaGen* l, int (*cb)(void*,void*), void* dado)
{
  ListaGen* p;
  for (p=l; p!=NULL; p=p->prox) {
     int r = cb(p->info,dado);
     if (r != 0)
        return r;
  }
  return 0;
}
```

Se usássemos essa versão de percorre para implementar as funções discutidas, teríamos de fazer as *callbacks* retornarem zero. O fato de permitir que ela retorne um valor possibilita fazer a busca de um ponto sem precisar continuar percorrendo os elementos após o ponto ser encontrado. Nossa *callback* recebe as coordenadas que buscamos como dado adicional e tem 1 como valor de retorno caso o ponto visitado tenha as mesmas coordenadas (o teste de igualdade das coordenadas é feito dentro de um intervalo de tolerância; podemos, por exemplo, fazer TOL valer 1e-5):

```
static int igualdade (void* info, void* dado)
{
  Ponto* p = (Ponto*)info;
  Ponto* q = (Ponto*)dado;
  if (fabs(p->x-p->x)<TOL && fabs(p->y-q->y)<TOL)
     return 1;
  else
     return 0;
}
```

Com isso, uma função do cliente para verificar a ocorrência das coordenadas (x,y) na estrutura é exemplificada por:

```
static int pertence (ListaGen* l, float x, float y)
{
  Ponto q;
  q.x = x; q.y = y;
  return lgen_percorre(l,igualdade,&q);
}
```

Essas técnicas de programação que utilizam *callbacks* são muito empregadas em programação, pois permitem esconder do cliente a forma como os elementos armazenados estão estruturados internamente. O cliente pode visitar e manipular todas as informações armazenadas, independente da estrutura de dados utilizada. Para o caso de uma lista simplesmente encadeada, o leitor pode questionar a real utilidade de implementar estruturas genéricas e funções que utilizam *callbacks*. No entanto, em estruturas mais sofisticadas, essa generalização é muito útil, pois só precisamos implementar a estrutura de dados uma única vez. Como veremos nos Capítulos 16 e 17, algoritmos genéricos implementados pela biblioteca padrão de C também fazem uso de *callbacks* para poderem ser independentes do tipo dos dados manipulados.

Exercícios

Os exercícios apresentados a seguir sugerem a implementação de diferentes funções. Para cada uma delas, o programador deve construir um programa (função `main`) para testar sua implementação.

1. Tipos abstratos de dados

1.1. Acrescente novas operações ao TAD ponto, como soma e subtração de pontos.

1.2. Acrescente novas operações ao TAD ponto, de forma que seja possível obter uma representação do ponto em coordenadas polares.

1.3. Use apenas as operações definidas pelo TAD matriz e implemente uma função que, dada uma matriz, crie dinamicamente a matriz transposta correspondente.

2. Listas encadeadas

2.1. Implemente uma função que tenha como valor de retorno o comprimento de uma lista encadeada, isto é, que calcule o número de nós de uma lista. Essa função deve obedecer ao protótipo:

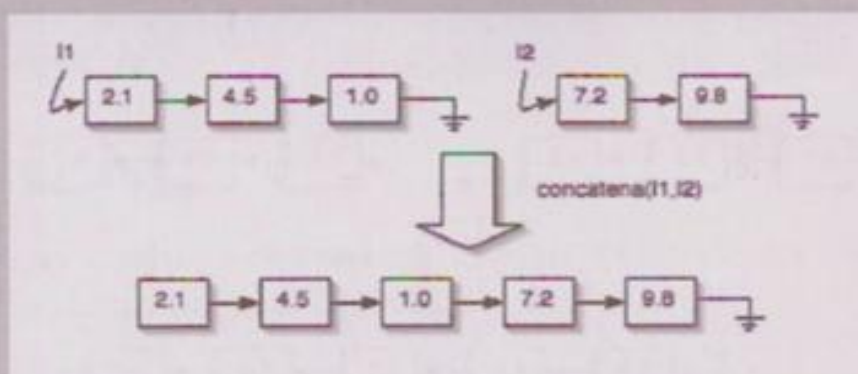
```
int comprimento (Lista* l);
```

2.2. Considere listas encadeadas de valores inteiros e implemente uma função para retornar o número de nós da lista que possuem o campo `info` com valores maiores do que `n`. Essa função deve obedecer ao protótipo:

```
int maiores (Lista* l, int n);
```

2.3. Implemente uma função que tenha como valor de retorno o ponteiro para o último nó de uma lista encadeada. Essa função deve obedecer ao protótipo:

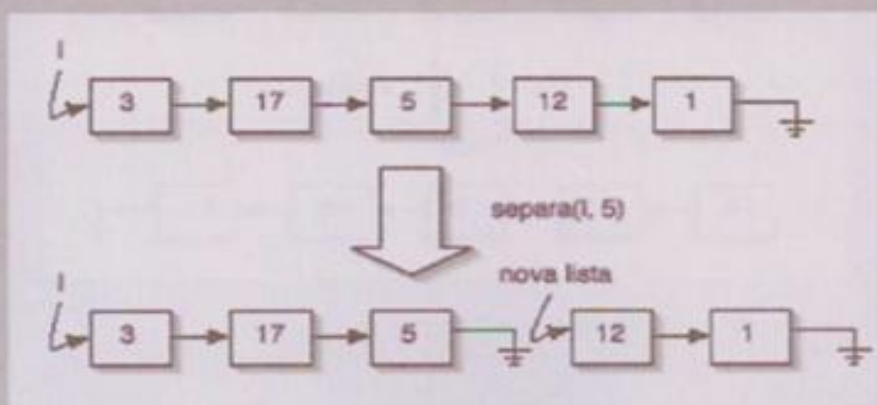
```
Lista* ultimo (Lista* l);
```

2.4. Implemente uma função que receba duas listas encadeadas de valores reais e retorne a lista resultante da concatenação das duas listas recebidas como parâmetros, isto é, após a concatenação, o último elemento da primeira lista deve apontar para o primeiro elemento da segunda lista, conforme ilustrado a seguir:

Essa função deve obedecer ao protótipo:

```
Lista* concatena (Lista* l1, Lista* l2);
```

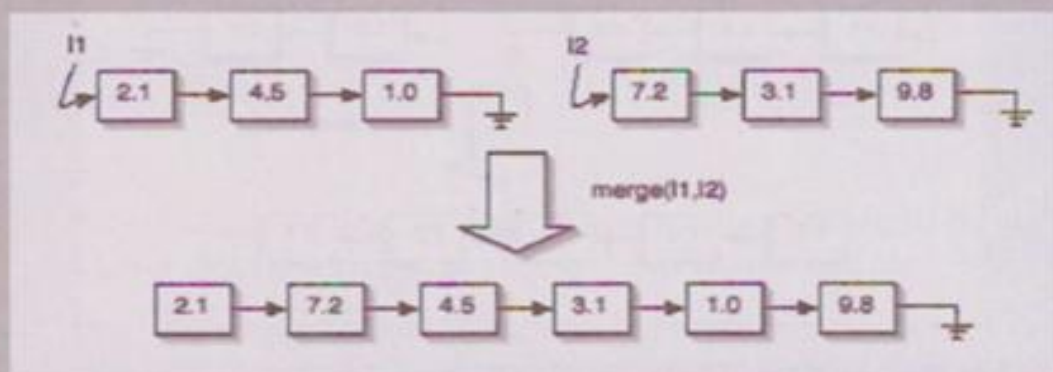
2.5. Considere listas de valores inteiros e implemente uma função que receba como parâmetros uma lista encadeada e um valor inteiro `n`, retire da lista todas as ocorrências de `n` e retorne a lista resultante. Essa função deve obedecer ao protótipo:

```
Lista* retira_n (Lista* l, int n);
```

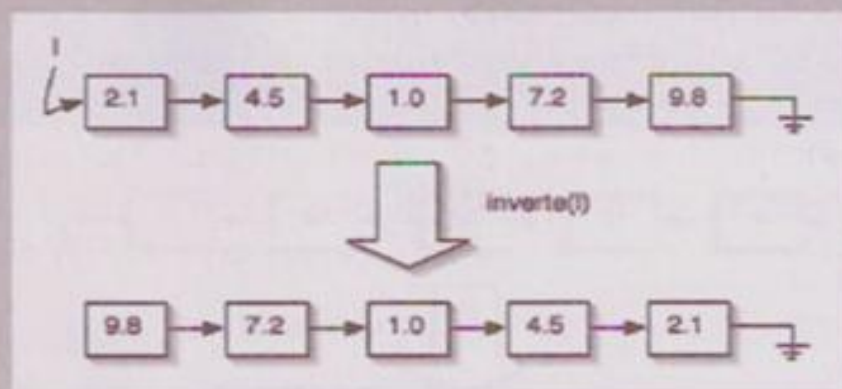
2.6. Considere listas de valores inteiros e implemente uma função que receba como parâmetro uma lista encadeada e um valor inteiro `n` e divida a lista em duas, de forma à segunda lista começar no primeiro nó logo após a primeira ocorrência de `n` na lista original. A figura a seguir ilustra essa separação:

Essa função deve obedecer ao protótipo:

```
Lista* separa (Lista* l, int n);
```

A função deve retornar um ponteiro para a segunda subdivisão da lista original, enquanto l deve continuar apontando para o primeiro elemento da primeira subdivisão da lista.

2.7. Implemente uma função que construa uma nova lista com a intercalação dos nós de outras duas listas passadas como parâmetros. Essa função deve retornar a lista resultante, conforme ilustrado a seguir:

Essa função deve obedecer ao protótipo:

```
Lista* merge (Lista* l1, Lista* l2);
```

2.8. Implemente uma função que receba como parâmetro uma lista encadeada e inverta o encadeamento de seus nós, retornando a lista resultante. Após a execução dessa função, cada nó da lista vai estar apontando para o nó que originalmente era seu antecessor, e o último nó da lista passará a ser o primeiro nó da lista invertida, conforme ilustrado a seguir:

Essa função deve obedecer ao protótipo:

```
Lista* inverte (Lista* l);
```

2.9. Considere listas que armazenam cadeias de caracteres e implemente uma função para testar se duas listas passadas como parâmetros são iguais. Essa função deve obedecer ao protótipo:

```
int igual (Lista* l1, Lista* l2);
```

2.10. Considere listas que armazenam cadeias de caracteres e implemente uma função para criar uma cópia de uma lista encadeada. Essa função deve obedecer ao protótipo:

```
Lista* copia (Lista* l);
```

2.11. Implemente funções para inserir e retirar um elemento de uma lista circular duplamente encadeada.

3. Pilhas e filas

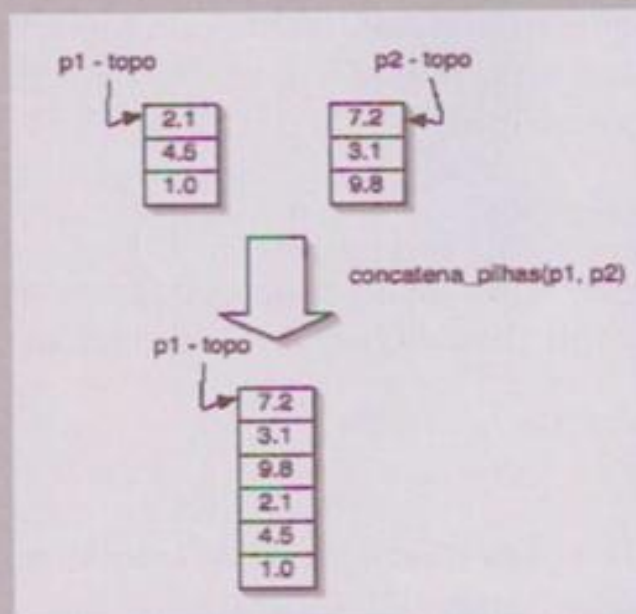
3.1. Considere a existência de um tipo abstrato `Pilha` de números reais, cuja interface está definida no arquivo *pilha.h* da seguinte forma:

```
typedef struct pilha Pilha;
Pilha* pilha_cria(void);
void   pilha_push (Pilha* p, float v);
float  pilha_pop (Pilha* p);
int    pilha_vazia (Pilha* p);
void   pilha_libera (Pilha* p);
```

Sem conhecer a representação interna desse tipo abstrato e de posse apenas das funções declaradas no arquivo de interface:

- a) Implemente uma função que receba uma pilha como parâmetro e retorne o valor armazenado em seu topo, restaurando o conteúdo da pilha. Essa função deve obedecer ao protótipo:

  ```
  float topo (Pilha* p);
  ```

- b) Implemente uma função que receba duas pilhas, `p1` e `p2`, e passe todos os elementos da pilha `p2` para o topo da pilha `p1`. A figura a seguir ilustra essa concatenação de pilhas:

Note que, ao final dessa função, a pilha p2 vai estar vazia, e a pilha p1 conterá todos os elementos das duas pilhas. Essa função deve obedecer ao protótipo:

```
void concatena_pilhas (Pilha* p1, Pilha* p2);
```

Essa função pode ser implementada mais facilmente por meio de uma solução recursiva ou de outra variável pilha auxiliar para fazer a transferência dos elementos entre as duas pilhas.

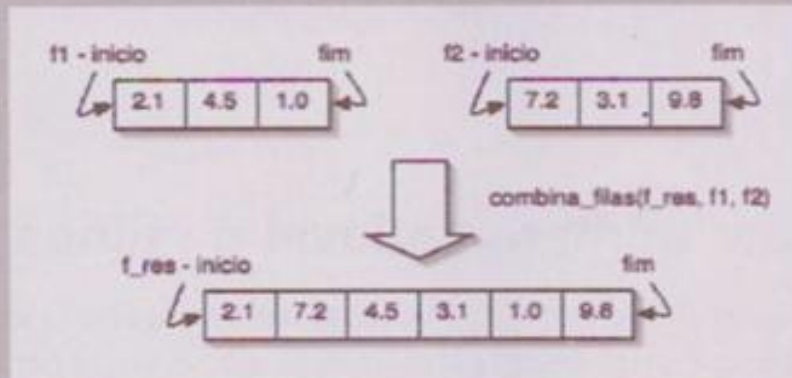
- c) Implemente uma função que receba uma pilha como parâmetro e retorne como resultado uma cópia dessa pilha. Essa função deve obedecer ao protótipo:

```
Pilha* copia_pilha (Pilha* p);
```

Ao final da função copia_pilha, a pilha p recebida como parâmetro deve ter seu conteúdo original. Essa função pode ser implementada mais facilmente com uma solução recursiva ou utilizando outra variável pilha auxiliar.

3.2. Considere a existência de um tipo abstrato Fila de números reais, cuja interface está definida no arquivo *fila.h* da seguinte forma:

```
typedef struct fila Fila;
Fila* fila_cria(void);
void  fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int   fila_vazia (Fila* f);
void  fila_libera (Fila* f);
```

Sem conhecer a representação interna desse tipo abstrato e usando apenas as funções declaradas no arquivo de interface, implemente uma função que receba três filas, `f_res`, `f1` e `f2`, e transfira alternadamente os elementos de `f1` e `f2` para `f_res`, conforme ilustrado a seguir:

Note que, ao final dessa função, as filas `f1` e `f2` vão estar vazias, e a fila `f_res` vai conter todos os valores originalmente em `f1` e `f2` (inicialmente `f_res` pode ou não estar vazia). Essa função deve obedecer ao protótipo:

```
void combina_filas (Fila* f_res, Fila* f1, Fila* f2);
```

3.3. Estenda a funcionalidade da calculadora pós-fixada que usa uma pilha de valores reais incluindo novos operadores unários e binários (sugestão: ~ como menos unário, # como raiz quadrada, ^ como exponenciação).

3.4. Implemente uma calculadora pós-fixada para operar sobre vetores do espaço 3D.

4. Árvores binárias

4.1. Considere estruturas de árvores binárias que armazenam valores inteiros e implemente uma função que, dada uma árvore, retorne a quantidade de nós que guardam números pares. Essa função deve obedecer ao protótipo:

```
int pares (Arv* a);
```

4.2. Implemente uma função que retorne a quantidade de folhas de uma árvore binária. Essa função deve obedecer ao protótipo:

```
int folhas (Arv* a);
```

4.3. Implemente uma função que retorne a quantidade de nós de uma árvore binária que possuem apenas um filho. Essa função deve obedecer ao protótipo:

```
int um_filho (Arv* a);
```

4.4. Implemente uma função que compare se duas árvores binárias são iguais. Essa função deve obedecer ao protótipo:

```
Arv* igual (Arv* a, Arv* b);
```

4.5. Implemente uma função que crie uma cópia de uma árvore binária. Essa função deve obedecer ao protótipo:

```
Arv* copia (Arv*a);
```

5. Árvores com número variável de filhos

5.1. Considere estruturas de árvores com número variável de filhos que armazenam valores inteiros e implemente uma função que, dada uma árvore, retorne a quantidade de nós que guardam números pares. Essa função deve obedecer ao protótipo:

```
int pares (ArvVar* a);
```

5.2. Implemente uma função que retorne a quantidade de folhas de uma árvore com número variável de filhos. Essa função deve obedecer ao protótipo:

```
int folhas (ArvVar* a);
```

5.3. Considere estruturas de árvores com número variável de filhos implemente uma função que retorne a quantidade de nós e com apenas um filho. Essa função deve obedecer ao protótipo:

```
int um_filho (ArvVar* a);
```

5.4. Implemente uma função que compare se duas árvores são iguais. Essa função deve obedecer ao protótipo:

```
ArvVar* igual (ArvVar* a, ArvVar* b);
```

5.5. Implemente uma função que crie uma cópia de uma árvore. Essa função deve obedecer ao protótipo:

```
ArvVar* copia (ArvVar*a);
```

PARTE III

Ordenação e busca

Nesta terceira parte do livro, focamos a discussão na implementação de dois tipos de algoritmos amplamente utilizados na elaboração de programas: ordenação e busca. Em diversas aplicações, os dados devem ser armazenados segundo uma determinada ordem, pois muitos algoritmos podem explorar essa ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional. O algoritmo de busca, por exemplo, pode tirar proveito da ordenação dos dados. A operação de busca é tão freqüente em aplicações computacionais que diversas estruturas de dados são projetadas especificamente para oferecer suporte eficiente a essa operação.

Nos capítulos anteriores, exibimos as estruturas de dados utilizadas para a organização de informações na memória do computador. O primeiro capítulo desta terceira parte, Capítulo 15, discute algumas técnicas para que possamos salvar e recuperar informações em arquivos de maneira estruturada. A seguir, no Capítulo 16, são introduzidos dois algoritmos para a ordenação de informações armazenadas em vetores e discutimos, em detalhes, técnicas de programação para a implementação de algoritmos genéricos, isto é, algoritmos que independem do tipo de informação que está sendo processada.

O Capítulo 17 descreve algoritmos de busca em vetores e aborda a estrutura de árvore binária que oferece suporte adequado a operações de busca. Por fim, no Capítulo 18, são mostradas as estruturas conhecidas como tabelas de dispersão, projetadas especificamente para realizar buscas de maneira extremamente eficiente.

15

Arquivos

Neste capítulo, apresentaremos alguns conceitos básicos sobre arquivos e alguns detalhes da forma de tratamento de arquivos em disco na linguagem C. A finalidade desta apresentação é discutir formas variadas para salvar (e recuperar) informações em arquivos. Com isso, será possível implementar funções para salvar (e recuperar) as informações armazenadas nas estruturas de dados discutidas.

Um arquivo em disco representa um elemento de informação do dispositivo de memória secundária. A memória secundária (disco) difere da memória principal em diversos aspectos. As duas diferenças mais relevantes são: eficiência e persistência. Enquanto o acesso a dados armazenados na memória principal é muito eficiente do ponto de vista de desempenho computacional, o acesso a informações armazenadas em disco é, em geral, extremamente ineficiente. Para contornar essa situação, os sistemas operacionais trabalham com *buffers*, que representam áreas da memória principal usadas como meio de transferência das informações de/para o disco. Normalmente, trechos maiores (alguns kbytes) são lidos e armazenados no *buffer* a cada acesso ao dispositivo. Dessa forma, uma subseqüente leitura de dados do arquivo, por exemplo, possivelmente não precisará acessar o disco, pois o dado requisitado pode já se encontrar no *buffer*. Os detalhes de como esses acessos se realizam dependem das características do dispositivo e do sistema operacional utilizado.

A outra grande diferença entre memória principal e secundária (disco) consiste no fato de as informações em disco serem persistentes, geralmente sendo lidas por programas e pessoas diferentes das que escreveram, o que torna mais prático atribuir nomes aos elementos de informação armazenados no disco (em vez de endereços), falando assim em arquivos e diretórios (pastas). Cada arquivo é identificado por seu nome e pelo diretório em que se encontra armazenado em uma determinada unidade de disco. Os nomes dos arquivos são, em geral, com-

postos pelo nome em si seguido de uma extensão. A extensão pode ser usada para identificar a natureza da informação armazenada no arquivo ou para identificar o programa que gerou (e é capaz de interpretar) o arquivo. Assim, a extensão ".c" é usada para identificar arquivos que têm códigos-fontes da linguagem C, e a extensão ".doc" é, no sistema operacional Windows®, usada para identificar arquivos gerados pelo editor Word da Microsoft®.

Um arquivo pode ser visto de duas maneiras, na maioria dos sistemas operacionais: em "modo texto", como um texto composto de uma seqüência de caracteres, ou em "modo binário", como uma seqüência de bytes (números binários). Podemos optar por salvar (e recuperar) informações em disco em um dos dois modos, texto ou binário. Uma vantagem do arquivo texto é que pode ser lido por uma pessoa e editado com editores de textos convencionais. Em contrapartida, com o uso de um arquivo binário é possível salvar (e recuperar) grandes quantidades de informação de forma mais eficiente. O sistema operacional pode tratar arquivos "texto" de maneira diferente da utilizada para tratar arquivos "binários". Em casos especiais, pode ser interessante tratar arquivos de um tipo como se fossem do outro, desde que tomados os cuidados apropriados.

Para minimizar a dificuldade na manipulação dos arquivos, os sistemas operacionais oferecem um conjunto de serviços para ler e escrever informações em disco. A linguagem C disponibiliza esses serviços para o programador por meio de um conjunto de funções. Os principais serviços que nos interessam são:

- abertura de arquivos: o sistema operacional encontra o arquivo com o nome dado e prepara o buffer na memória;
- leitura do arquivo: o sistema operacional recupera o trecho solicitado do arquivo. Como o buffer contém parte da informação do arquivo, parte ou toda a informação solicitada pode vir dele;
- escrita no arquivo: o sistema operacional acrescenta ou altera o conteúdo do arquivo. A alteração no conteúdo do arquivo é feita inicialmente no buffer para depois ser transferida para o disco;
- fechamento de arquivo: toda a informação contida no buffer é atualizada no disco e a área do buffer utilizada na memória é liberada.

Uma das informações mantidas pelo sistema operacional é um *cursor* que indica a posição de trabalho no arquivo. Para leitura, esse cursor percorre a seqüência de informação existente no arquivo, do início até o fim, conforme os dados vão sendo recuperados (lidos) para a memória. Para escrita, normalmente, os dados são acrescentados quando o cursor se encontra no fim do arquivo.

Nas seções subseqüentes, vamos apresentar as funções mais utilizadas em C para acessar arquivos e discutir diferentes estratégias para tratá-los. Todas as funções da biblioteca padrão de C que manipulam arquivos encontram-se na biblioteca de entrada e saída, com interface em *stdio.h*.

Funções para abrir e fechar arquivos

A função básica para abrir um arquivo é `fopen`:[1]

```
FILE* fopen (char* nome_arquivo, char* modo);
```

`FILE` é um tipo definido pela biblioteca padrão que representa uma abstração do arquivo. Quando abrimos um arquivo, a função tem como valor de retorno um ponteiro para o tipo `FILE`, e todas as operações subseqüentes nesse arquivo receberão esse endereço como parâmetro de entrada. Se o arquivo não puder ser aberto, a função tem como retorno o valor `NULL`.

Devemos passar o nome do arquivo a ser aberto. O nome do arquivo pode ser relativo, e o sistema o procura a partir do diretório corrente (diretório de trabalho do programa), ou pode ser absoluto, e para tanto especificamos o nome completo do arquivo, o que inclui os diretórios, desde o diretório raiz.

Existem diferentes modos de abertura de um arquivo. Podemos abrir um arquivo para leitura ou para escrita e devemos especificar se o arquivo será aberto em modo texto ou em modo binário. O parâmetro `modo` da função `fopen` é uma cadeia de caracteres em que se espera a ocorrência de caracteres que identificam o modo de abertura. Os caracteres interpretados no modo são:

`r`	*read*	Indica modo para leitura;
`w`	*write*	Indica modo para escrita;
`a`	*append*	Indica modo para escrita ao final do existente;
`t`	*text*	Indica modo texto;
`b`	*binary*	Indica modo binário.

Se o arquivo já existe e solicitamos a sua abertura para escrita com modo `w`, o arquivo é destruído e um novo, inicialmente vazio, é criado. Quando solicitamos com modo `a`, o mesmo é preservado, e novos conteúdos podem ser escritos no seu fim. Com ambos os modos, se o arquivo não existe, um novo é criado. Se solicitarmos a abertura de um arquivo para leitura, ele já deve existir; caso contrário a função falha e tem como retorno o valor `NULL`. A função também tem `NULL` como valor de retorno se tentarmos abrir um arquivo para escrita em uma área (diretório) na qual não temos acesso de escrita. Se quisermos abrir um arquivo para simultaneamente ler e escrever, acrescentamos o caractere `+` no modo de abertura. Assim, `r+` indica leitura e escrita em um arquivo já existente e `w+` indica leitura e escrita em um novo arquivo.

Os modos `b` e `t` podem ser combinados com os demais. Mais detalhes podem ser encontrados nos manuais da linguagem C. Em geral, quando abrimos um arquivo, testamos o sucesso da abertura antes de qualquer outra operação, como, por exemplo:

---

[1]A rigor, os parâmetros do tipo cadeias de caracteres são declarados com o modificador `const`.

```
...
FILE* fp;
fp = fopen("entrada.txt","rt");
if (fp == NULL) {
  printf("Erro na abertura do arquivo!\n");
  exit(1);
}
...
```

Nesse fragmento de código, solicitamos a abertura do arquivo de nome *entrada.txt* para leitura em modo texto. Em seguida, testamos se a abertura do arquivo foi realizada com sucesso.

Após ler/escrever as informações de um arquivo, devemos fechá-lo. Para isso, devemos usar a função `fclose`, a qual espera como parâmetro o ponteiro do arquivo que se deseja fechar. O protótipo da função é:

```
int fclose (FILE* fp);
```

O valor de retorno dessa função é zero, se o arquivo for fechado com sucesso, ou a constante `EOF` (definida pela biblioteca), que indica a ocorrência de um erro.

Arquivos em modo texto

Nesta seção, descreveremos as principais funções para manipular arquivos em modo texto. Também discutiremos algumas estratégias para a organização de dados em arquivos.

Funções para ler dados

A principal função de C para a leitura de dados em arquivos em modo texto é a função `fscanf`, similar à função `scanf` que temos usado para capturar valores inseridos via teclado. No caso da `fscanf`, os dados são capturados de um arquivo previamente aberto para leitura. A cada leitura, os dados correspondentes são transferidos para a memória, e o cursor do arquivo avança, passando a apontar para o próximo dado do arquivo (que pode ser capturado numa leitura subseqüente). O protótipo da função `fscanf` é:

```
int fscanf (FILE* fp, char* formato, ...);
```

Conforme pode ser observado, o primeiro parâmetro deve ser o ponteiro para o arquivo do qual os dados serão lidos. Os demais parâmetros são os já discutidos para a função `scanf`: o formato e a lista de endereços de variáveis que armazenarão os valores lidos. Assim como a função `scanf`, a função `fscanf` também tem como valor de retorno o número de dados lidos com sucesso.

Outra função de leitura muito usada em modo texto é a função `fgetc` que, dado o ponteiro do arquivo, captura o próximo caractere do arquivo (e o cursor avança para o próximo caractere). O protótipo dessa função é:

```
int fgetc (FILE* fp);
```

Apesar de o tipo do valor de retorno ser `int`, o valor retornado é o código do caractere lido. Se o fim do arquivo for alcançado, a constante `EOF` (*end of file*) é retornada.

Outra função muito utilizada para ler linhas de um arquivo é a função `fgets`. Ela recebe como parâmetros três valores: a cadeia de caracteres que armazenará o conteúdo lido do arquivo, o número máximo de caracteres que deve ser lido e o ponteiro do arquivo. O protótipo da função é:

```
char* fgets (char* s, int n, FILE* fp);
```

A função lê do arquivo uma seqüência de caracteres, até que um caractere `'\n'` seja encontrado ou o máximo de caracteres especificado seja alcançado. A especificação de um número máximo de caracteres é importante para evitar invadir memória quando a linha do arquivo for maior do que supúnhamos. Assim, se dimensionarmos nossa cadeia de caracteres, a qual receberá o conteúdo da linha lida, com 121 caracteres, passaremos esse valor para a função, que lerá no máximo 120 caracteres, pois o último será ocupado pelo finalizador de *string* – o caractere `'\0'`. O valor de retorno dessa função é o ponteiro da própria cadeia de caracteres passada como parâmetro ou `NULL` no caso de ocorrer erro de leitura (por exemplo, quando alcançar o final do arquivo).

É importante salientar que a informação lida é sempre a informação apontada pelo cursor do arquivo. Quando abrimos um arquivo para leitura, esse cursor é automaticamente posicionado no início do arquivo. A cada leitura, o cursor avança e passa a apontar para a posição imediatamente após a informação lida. Assim, em uma próxima leitura, captura-se a próxima informação do arquivo.

Funções para escrever dados

Dentre as funções existentes para escrever (salvar) dados em um arquivo texto, vamos considerar as duas mais freqüentemente utilizadas: `fprintf` e `fputc`, análogas, mas para escrita, às funções que vimos para leitura.

A função `fprintf` é similar à função `printf` que temos usado para imprimir dados na saída padrão – em geral, o monitor. A diferença consiste na presença do parâmetro que indica o arquivo para o qual o dado será salvo. O valor de retorno dessa função representa o número de bytes escritos no arquivo. O protótipo da função é dado por:

```
int fprintf(FILE* fp, char* formato, ...);
```

A função fputc escreve um caractere no arquivo. O protótipo é:

```
int fputc (int c, FILE* fp);
```

No primeiro parâmetro, especificamos o código do caractere que queremos escrever (salvar). O valor de retorno dessa função é o próprio caractere escrito, ou EOF se ocorrer um erro na escrita.

Estruturação de dados em arquivos textos

Existem diferentes maneiras de estruturar os dados em arquivos em modo texto, bem como de capturar as informações contidas neles. A forma de estruturar e a forma de tratar as informações dependem da aplicação. A seguir, apresentaremos três modos para representar e acessar dados armazenados em arquivos: caractere a caractere, linha a linha e com palavras-chaves.

Acesso caractere a caractere

Para exemplificar o acesso caractere a caractere, vamos discutir duas aplicações simples. Inicialmente, vamos considerar o desenvolvimento de um programa que conta o número de linhas de um determinado arquivo (para simplificar, vamos supor um arquivo fixo, com o nome "entrada.txt"). Para calcular o número de linhas do arquivo, podemos ler, caractere a caractere, todo o conteúdo do arquivo, e contar o número de ocorrências do caractere que indica mudança de linha, isto é, o número de ocorrências de '\n'.

```
/* Conta número de linhas de um arquivo */

#include <stdio.h>

int main (void)
{
  int c;
  int nlinhas = 0;      /* contador do número de linhas */
  FILE *fp;

  /* abre arquivo para leitura */
  fp = fopen("entrada.txt","rt");
  if (fp==NULL) {
     printf("Não foi possível abrir arquivo.\n");
     return 1;
  }
```

```
   /* lê caractere a caractere */
   while ((c = fgetc(fp)) != EOF) {
      if (c == '\n')
         nlinhas++;
   }

   /* fecha arquivo */
   fclose(fp);

   /* exibe resultado na tela */
   printf("Número de linhas = %d\n", nlinhas);

   return 0;
}
```

Nesse programa, como capturamos caractere a caractere, usamos a função `fgetc`, declarando a variável `c` como sendo do tipo `int` (pois `fgetc` retorna um `int`). Como alternativa, podemos reescrever o código com a função `fscanf` para fazer a leitura dos caracteres:

```
   ...
   char c;
   ...
   while (fscanf("%c",&c)==1) {
      if (c == '\n')
         nlinhas++;
   }
   ...
```

Em um segundo exemplo, vamos considerar o desenvolvimento de um programa que lê o conteúdo do arquivo e cria um arquivo com o mesmo conteúdo, mas com todas as letras minúsculas convertidas para maiúsculas. Os nomes dos arquivos serão fornecidos, via teclado, pelo usuário. Uma possível implementação desse programa é mostrada a seguir:

```
/* Converte arquivo para maiúsculas */

#include <stdio.h>
#include <ctype.h>    /* função toupper */

int main (void)
{
   int c;
   char entrada[121];    /* armazena nome do arquivo de entrada */
   char saida[121];      /* armazena nome do arquivo de saída */
   FILE* e;              /* ponteiro do arquivo de entrada */
   FILE* s;              /* ponteiro do arquivo de saída */
```

```
  /* pede ao usuário os nomes dos arquivos */
  printf("Digite o nome do arquivo de entrada: ");
  scanf("%120s",entrada);
  printf("Digite o nome do arquivo de saída: ");
  scanf("%120s",saida);

  /* abre arquivos para leitura e para escrita */
  e = fopen(entrada,"rt");
  if (e == NULL) {
     printf("Não foi possível abrir arquivo de entrada.\n");
     return 1;
  }
  s = fopen(saida,"wt");
  if (s == NULL) {
     printf("Não foi possível abrir arquivo de saída.\n");
     fclose(e);
     return 1;
  }

  /* lê da entrada e escreve na saída */
  while ((c = fgetc(e)) != EOF)
     fputc(toupper(c),s);

  /* fecha arquivos */
  fclose(e);
  fclose(s);

  return 0;
}
```

Novamente, poderíamos ter usado as funções `fscanf` e `fprintf` para a leitura e a escrita dos caracteres.

Por fim, vale salientar que a linguagem C oferece a função `ungetc`, a qual permite "devolver" o último caractere lido. Se devolvermos um caractere, ele mesmo será capturado em uma próxima leitura. Essa função é muito útil quando nossa aplicação precisa "ver", sem avançar com o cursor, qual é a informação seguinte e então decidir que procedimento deve ser adotado. Nós usamos essa função quando apresentamos o exemplo da calculadora pós-fixada, no Capítulo 11.

Acesso linha a linha

Em diversas aplicações, é mais adequado tratar o conteúdo do arquivo linha a linha. Um caso simples que podemos mostrar consiste em procurar a ocorrência de uma subcadeia de caracteres dentro de um arquivo (análogo ao que é feito pelo utilitário `grep` dos sistemas Unix). Se a subcadeia for encontrada, apresentamos como saída o número da linha da primeira ocorrência.

Para implementar esse programa, vamos utilizar a função `strstr`, que procura a ocorrência de uma subcadeia em uma cadeia de caracteres maior. A função tem como valor de retorno o endereço da primeira ocorrência ou `NULL`, se a subcadeia não for encontrada. O protótipo dessa função é:

```
char* strstr (char* s, char* sub);
```

A nossa implementação consistirá em ler, linha a linha, o conteúdo do arquivo, contando o número da linha. Para cada linha, verificamos a ocorrência da subcadeia e interrompemos a leitura em caso afirmativo.

```
/* Procura ocorrência de subcadeia no arquivo */

#include <stdio.h>
#include <string.h>    /* função strstr */

int main (void)
{
  int n = 0;               /* número da linha corrente */
  int achou = 0;           /* indica se achou subcadeia */
  char entrada[121];       /* armazena nome do arquivo de entrada */
  char subcadeia[121];     /* armazena subcadeia */
  char linha[121];         /* armazena cada linha do arquivo */
  FILE* fp;                /* ponteiro do arquivo de entrada */

  /* pede ao usuário o nome do arquivo e a subcadeia */
  printf("Digite o nome do arquivo de entrada: ");
  scanf("%120s",entrada);
  printf("Digite a subcadeia: ");
  scanf("%120s",subcadeia);

  /* abre arquivos para leitura */
  fp = fopen(entrada,"rt");
  if (fp == NULL) {
     printf("Não foi possível abrir arquivo de entrada.\n");
     return 1;
  }

  /* lê linha a linha */
  while (fgets(linha,121,fp) != NULL) {
     n++;
     if (strstr(linha,subcadeia) != NULL) {
        achou = 1;
        break;
     }
  }
```

```
    /* fecha arquivo */
    fclose(fp);

    /* exibe saída */
    if (achou)
        printf("Achou na linha %d.\n", n);
    else
        printf("Não achou.");

    return 0;
}
```

Como segundo exemplo de arquivos manipulados linha a linha, podemos citar o caso em que salvamos os dados com formatação por linha. Para exemplificar, vamos considerar que queremos salvar as informações da lista de figuras geométricas discutidas no Capítulo 10. A lista continha retângulos, triângulos e círculos.

Para salvar essas informações em um arquivo, temos de escolher um formato apropriado, o qual nos permita posteriormente recuperar a informação salva. Para exemplificar um formato válido, vamos adotar uma formatação por linha: em cada linha salvamos um caractere que indica o tipo da figura (r, t ou c), seguido dos parâmetros que definem a figura: base e altura para os retângulos e triângulos ou raio para os círculos. Para enriquecer o formato, podemos considerar que as linhas iniciadas com o caractere # representam comentários e devem ser desconsideradas na leitura. Por fim, linhas em branco são permitidas e desprezadas. Um exemplo do conteúdo de um arquivo com esse formato é apresentado na Figura 15.1 (note a presença de linhas em branco e linhas que são comentários).

```
# Lista de figuras geométricas


r 2.0 1.2

c 5.8

# t 1.23 12

t 4 1.02


c 5.1
```

Figura 15.1 *Exemplo de formatação por linha.*

Para recuperar as informações contidas em um arquivo com esse formato, podemos ler do arquivo cada uma das linhas e depois ler os dados contidos na linha. Para tanto, precisamos apresentar uma função adicional muito útil. Trata-se da função que permite ler dados de uma cadeia de caracteres. A função `sscanf` é similar às funções `scanf` e `fscanf`, mas captura os valores armazenados em uma string. O protótipo dessa função é:

```
int sscanf (char* s, char* formato, ...);
```

A primeira cadeia de caracteres passada como parâmetro representa a string da qual os dados serão lidos. Com essa função, é possível ler uma linha de um arquivo e depois ler as informações contidas na linha. (Analogamente, existe a função `sprintf`, que permite escrever dados formatados numa string).

Faremos a interpretação do arquivo da seguinte forma: para cada linha lida do arquivo, tentaremos ler do conteúdo da linha um caractere (desprezando eventuais caracteres brancos iniciais) seguido de dois números reais. Se nenhum dado for lido com sucesso, significa que temos uma linha vazia e devemos desprezá-la. Se pelo menos um dado (no caso, o caractere) for lido com sucesso, podemos interpretar o tipo da figura geométrica armazenada na linha ou detectar a ocorrência de um comentário. Se for um retângulo ou um triângulo, os dois valores reais também deverão ter sido lidos com sucesso. Se for um círculo, apenas um valor real deverá ter sido lido com sucesso. O fragmento de código a seguir ilustra essa implementação. Supõe-se que `fp` representa um ponteiro para um arquivo com formato válido aberto para leitura, em modo texto.

```
char c;
float v1, v2;
FILE* fp;
char linha[121];
...
while (fgets(linha,121,fp)) {
   int n = sscanf(linha," %c %f %f",&c,&v1,&v2);
   if (n>0) {
      switch(c) {
         case '#':
            /* desprezar linha de comentário */
         break;
         case 'r':
            if (n!=3) {
               /* tratar erro de formato do arquivo */
               ...
            }
            else {
               /* tratar retângulo: base = v1, altura = v2 */
               ...
            }
         break;
```

```
            case 't':
               if (n!=3) {
                  /* tratar erro de formato do arquivo */
                  ...
               }
               else {
                  /* tratar triângulo: base = v1, altura = v2 */
                  ...
               }
            break;
            case 'c':
               if (n!=2) {
                  /* tratar erro de formato do arquivo */
                  ...
               }
               else {
                  /* tratar círculo: raio = v1 */
                  ...
               }
            break;
            default:
               /* tratar erro de formato do arquivo */
               ...
            break;
         }
      }
   }
   ...
```

A rigor, para o formato descrito, não precisávamos fazer a interpretação do arquivo linha a linha. O arquivo poderia ter sido interpretado com a captura inicial de um caractere, que então indicaria a próxima informação a ser lida. No entanto, em algumas situações, a interpretação linha a linha ilustrada é a única forma possível. Para exemplificar, vamos considerar um arquivo que representa um conjunto de pontos no espaço 3D. Esses pontos podem ser dados pelas suas três coordenadas *x*, *y* e *z*. Um formato bastante flexível para esse arquivo considera que cada ponto é dado em uma linha e permite a omissão da terceira coordenada, se ela for igual a zero. Dessa forma, o formato atende também à descrição de pontos no espaço 2D. Um exemplo desse formato é ilustrado a seguir:

```
2.3   4.5  6.0
1.2  10.4
7.4   1.3  9.6
...
```

Para interpretar esse formato, devemos ler cada uma das linhas e tentar ler três valores reais de cada linha (aceitando o caso de apenas dois valores serem lidos com sucesso).

Acesso via palavras-chave

Quando os objetos em um arquivo têm descrições de tamanhos variados, é comum adotarmos uma formatação com o uso de palavras-chave. Cada objeto é precedido por uma palavra-chave que o identifica. A interpretação desse tipo de arquivo pode ser feita com a leitura das palavras-chave e a interpretação da descrição do objeto correspondente. Para ilustrar, vamos considerar que, além de retângulos, triângulos e círculos, também temos polígonos quaisquer no nosso conjunto de figuras geométricas. Cada polígono pode ser descrito pelo número de vértices que o compõe, seguido das respectivas coordenadas desses vértices. A Figura 15.2 ilustra esse formato.

```
RETANGULO
b   h
TRIANGULO
b   h
CIRCULO
r
POLIGONO
n
x1  y1
x2  y2
...
xn  yn
```

Figura 15.2 *Formato com uso de palavras-chave.*

O fragmento de código a seguir ilustra a interpretação desse formato, em que fp representa o ponteiro para o arquivo aberto para leitura.

```
...
FILE* fp;
char palavra[121];
...
```

```
while (fscanf(fp,"%120s",palavra) == 1)
{
   if (strcmp(palavra,"RETANGULO")==0) {
      /* interpreta retângulo */
   ...
   }
   else if (strcmp(palavra,"TRIANGULO")==0) {
      /* interpreta triângulo */
   ...
   }
   else if (strcmp(palavra,"CIRCULO")==0) {
      /* interpreta círculo */
   ...
   }
   else if (strcmp(palavra,"POLIGONO")==0) {
      /* interpreta polígono */
   ...
   }
   else {
      /* trata erro de formato */
   ...
   }
}
```

Arquivos em modo binário

Os arquivos em modo binário servem para salvar (e depois recuperar) o conteúdo da memória principal diretamente no disco. A memória é escrita ao se copiar o conteúdo de cada byte da memória para o arquivo. Uma das grandes vantagens de usar arquivos binários é que podemos salvar (e recuperar) uma grande quantidade de dados de forma mais eficiente. Nesta seção, vamos apenas apresentar as funções básicas para a manipulação de arquivos binários.

Funções para salvar e recuperar

Para escrever (salvar) dados em arquivos binários, usamos a função `fwrite`. O protótipo dessa função pode ser simplificado por[1]:

```
int fwrite (void* p, int tam, int nelem, FILE* fp);
```

O primeiro parâmetro dessa função representa o endereço de memória cujo conteúdo se deseja salvar em arquivo. O parâmetro `tam` indica o tamanho, em bytes, de cada elemento, e o parâmetro `nelem` indica o número de elementos. Por fim, passa-se o ponteiro do arquivo binário para o qual o conteúdo da memória será copiado.

[1] A rigor, os tipos `int` são substituídos pelo tipo `size_t`, definido pela biblioteca padrão, sendo, em geral, sinônimo para um inteiro sem sinal (`unsigned int`).

A função para ler (recuperar) dados de arquivos binários é análoga, sendo que agora o conteúdo do disco é copiado para o endereço de memória passado como parâmetro. O protótipo da função pode ser dado por:

```
int fread (void* p, int tam, int nelem, FILE* fp);
```

Para exemplificar a utilização dessas funções, vamos considerar que uma aplicação tem um conjunto de pontos armazenados em um vetor. O tipo que define o ponto pode ser:

```
struct ponto {
  float x, y, z;
};
typedef struct ponto Ponto;
```

Uma função para salvar o conteúdo de um vetor de pontos pode receber como parâmetros o nome do arquivo, o número de pontos no vetor e o ponteiro para o vetor. Uma possível implementação dessa função é ilustrada a seguir:

```
void salva (char* arquivo, int n, Ponto* vet)
{
  FILE* fp = fopen(arquivo,"wb");
  if (fp==NULL) {
     printf("Erro na abertura do arquivo.\n");
     exit(1);
  }
  fwrite(vet,sizeof(Ponto),n,fp);
  fclose(fp);
}
```

A função para recuperar os dados salvos pode ser:

```
void carrega (char* arquivo, int n, Ponto* vet)
{
  FILE* fp = fopen(arquivo,"rb");
  if (fp==NULL) {
     printf("Erro na abertura do arquivo.\n");
     exit(1);
  }
  fread(vet,sizeof(Ponto),n,fp);
  fclose(fp);
}
```

Outra grande vantagem oferecida pelo uso de arquivos binários consiste na possibilidade de recuperar apenas parte da informação armazenada. Em um arquivo binário, nós, programadores, temos o controle de quantos bytes ocupa cada informação armazenada no arquivo. Com isso, podemos alterar a posição do cursor do arquivo, o que permite posicioná-lo para ler uma determinada informação. A função que permite movimentar o cursor do arquivo tem o seguinte protótipo:

```
int fseek (FILE* fp, long offset, int origem);
```

O primeiro parâmetro indica o arquivo no qual estamos reposicionando o cursor. O segundo parâmetro indica quantos bytes iremos avançar, e o terceiro parâmetro indica em relação a que posição estamos avançando o cursor: em relação à posição corrente (SEEK_CUR), em relação ao início do arquivo (SEEK_SET) ou em relação ao final do arquivo (SEEK_END).

Para exemplificar, vamos considerar a existência de um arquivo de pontos no espaço 3D salvo como exemplificado anteriormente. Vamos então escrever uma função que, dado um ponteiro para esse arquivo aberto para leitura, faça a captura do i-ésimo ponto armazenado. Uma possível implementação dessa função é mostrada a seguir:

```
Ponto le_ponto (FILE* fp, int i)
{
  Ponto p;
  fseek(fp,i*sizeof(Ponto),SEEK_SET);
  fread(&p,sizeof(Ponto),1,fp);
  return p;
}
```

16

Ordenação

Em diversas aplicações, os dados devem ser armazenados de acordo com uma determinada ordem. Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional. Para ordenar os dados, temos basicamente duas alternativas: ou inserimos os elementos na estrutura de dados respeitando a ordenação (dizemos que a ordenação é garantida por construção) ou, a partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos. Neste capítulo, discutiremos dois algoritmos de ordenação que podem ser empregados em aplicações computacionais.

Devido ao seu uso muito freqüente, é importante ter à disposição algoritmos de ordenação (*sorting*) eficientes em termos de tempo (devem ser rápidos) e em termos de espaço (devem ocupar pouca memória durante a execução). Vamos descrever os algoritmos de ordenação no seguinte cenário:

- a entrada é um vetor cujos elementos precisam ser ordenados;
- a saída é o mesmo vetor com seus elementos na ordem especificada.

Portanto, vamos discutir ordenação de vetores. Como veremos, os algoritmos de ordenação podem ser aplicados a qualquer informação, desde que exista uma ordem definida entre os elementos. Podemos, por exemplo, ordenar um vetor de valores inteiros que adote uma ordem crescente ou decrescente. Podemos também aplicar algoritmos de ordenação em vetores responsáveis por guardar informações mais complexas, por exemplo, um vetor que guarda os dados relativos a alunos de uma turma, com nome, número de matrícula etc. Nesse caso, a ordem entre os elementos tem de ser definida usando uma das informações do aluno como chave da ordenação: alunos ordenados pelo nome, alunos ordenados pelo número de matrícula etc.

Nos casos de informação complexa, raramente se encontra toda a informação relevante sobre os elementos do vetor no próprio vetor; em vez disso, cada componente pode conter apenas um ponteiro para a informação propriamente dita, que pode ficar em outra posição na memória. Assim, a ordenação pode ser feita sem a necessidade de mover grandes quantidades de informações para rearrumar as componentes do vetor na ordem correta. Para trocar a ordem entre dois elementos, apenas os ponteiros são trocados. Em muitos casos, devido ao grande volume, as informações podem ficar em um arquivo de disco, e o elemento do vetor ser apenas uma referência para a posição da informação nesse arquivo.

Neste capítulo, examinaremos os algoritmos de ordenação conhecidos como "ordenação bolha" (*bubble sort*) e "ordenação rápida" (*quick sort*), ou, mais precisamente, versões simplificadas desses algoritmos.

Ordenação bolha

O algoritmo de "ordenação bolha", ou "*bubble sort*", recebeu esse nome pela imagem pitoresca usada para descrevê-lo: os elementos maiores são mais leves e sobem como bolhas até suas posições corretas. A idéia fundamental é fazer uma série de comparações entre os elementos do vetor. Quando dois elementos estão fora de ordem, há uma inversão, e esses dois elementos são trocados de posição, ficando em ordem correta. Assim, o primeiro elemento é comparado com o segundo. Se uma inversão for encontrada, a troca é feita. Em seguida, independente de se houve ou não troca após a primeira comparação, o segundo elemento é comparado com o terceiro, e, caso uma inversão seja encontrada, a troca é feita. O processo continua até que o penúltimo elemento seja comparado com o último. Com esse processo, garante-se que o elemento de maior valor do vetor seja levado para a última posição. A ordenação continua, com o posicionamento do segundo maior elemento, do terceiro etc., até que todo o vetor esteja ordenado.

Para exemplificar, vamos considerar os elementos do vetor que queremos ordenar como valores inteiros. Assim, consideremos a ordenação do seguinte vetor:

```
25 48 37 12 57 86 33 92
```

Seguimos os passos indicados:

```
25 48 37 12 57 86 33 92   25x48
25 48 37 12 57 86 33 92   48x37 troca
25 37 48 12 57 86 33 92   48x12 troca
25 37 12 48 57 86 33 92   48x57
25 37 12 48 57 86 33 92   57x86
25 37 12 48 57 86 33 92   86x33 troca
```

```
25 37 12 48 57 33 86 92   86x92
25 37 12 48 57 33 86 92   final da primeira passada
```

Nesse ponto, o maior elemento, 92, já está na sua posição final.

```
25 37 12 48 57 33 86 92   25x37
25 37 12 48 57 33 86 92   37x12 troca
25 12 37 48 57 33 86 92   37x48
25 12 37 48 57 33 86 92   48x57
25 12 37 48 57 33 86 92   57x33 troca
25 12 37 48 33 57 86 92   57x86
25 12 37 48 33 57 86 92   final da segunda passada
```

Nesse ponto, o segundo maior elemento, 86, já está na sua posição final.

```
25 12 37 48 33 57 86 92   25x12 troca
12 25 37 48 33 57 86 92   25x37
12 25 37 48 33 57 86 92   37x48
12 25 37 48 33 57 86 92   48x33 troca
12 25 37 33 48 57 86 92   48x57
12 25 37 33 48 57 86 92   final da terceira passada
```

Idem para 57.

```
12 25 37 33 48 57 86 92   12x25
12 25 37 33 48 57 86 92   25x37
12 25 37 33 48 57 86 92   37x33 troca
12 25 33 37 48 57 86 92   37x48
12 25 33 37 48 57 86 92   final da quarta passada
```

Idem para 48.

```
12 25 33 37 48 57 86 92   12x25
12 25 33 37 48 57 86 92   25x33
12 25 33 37 48 57 86 92   33x37
12 25 33 37 48 57 86 92   final da quinta passada
```

Idem para 37.

```
12 25 33 37 48 57 86 92   12x25
12 25 33 37 48 57 86 92   25x33
12 25 33 37 48 57 86 92   final da sexta passada
```

Idem para 33.

```
12 25 33 37 48 57 86 92   12x25
12 25 33 37 48 57 86 92   final da sétima passada
```

Idem para 25 e, conseqüentemente, 12.

```
12 25 33 37 48 57 86 92    final da ordenação
```

A parte sabidamente já ordenada do vetor está sublinhada. Na realidade, após a troca de 37x33, o vetor se encontra totalmente ordenado, mas esse fato não é levado em consideração por essa versão do algoritmo.

Uma função que implementa esse algoritmo é apresentada a seguir. A função recebe como parâmetros o número de elementos e o ponteiro do primeiro elemento do vetor que se deseja ordenar. Vamos considerar a ordenação de um vetor de valores inteiros.

```
/* Ordenação bolha */
void bolha (int n, int* v)
{
  int i,j;
  for (i=n-1; i>=1; i--)
     for (j=0; j<i; j++)
        if (v[j]>v[j+1]) {
           int temp = v[j];        /* troca */
           v[j] = v[j+1];
           v[j+1] = temp;
        }
}
```

Uma função cliente para testar esse algoritmo pode ser dada por:

```
/* Testa algoritmo de ordenação bolha */
#include <stdio.h>

int main (void)
{
  int i;
  int v[8] = {25,48,37,12,57,86,33,92};
  bolha(8,v);
  printf("Vetor ordenado: ");
  for (i=0; i<8; i++)
     printf("%d ",v[i]);
  printf("\n");
  return 0;
}
```

Para evitar que o processo continue mesmo depois de o vetor estar ordenado, podemos interromper o processo quando houver uma passagem inteira sem trocas, usando uma variante do algoritmo apresentado acima:

```
/* Ordenação bolha (2a. versão) */
void bolha (int n, int* v)
```

```
{
    int i, j;
    for (i=n-1; i>0; i--) {
        int troca = 0;
        for (j=0; j<i; j++)
            if (v[j]>v[j+1]) {
                int temp = v[j];        /* troca */
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
            }
        if (troca == 0)                 /* não houve troca */
            return;
    }
}
```

A variável `troca` guarda o valor `0` (falso) quando uma passada do vetor (no `for` interno) se faz sem nenhuma troca.

O esforço computacional despendido pela ordenação de um vetor pode ser determinado pelo número de comparações, que serve também para estimar o número máximo de trocas possíveis de se realizar. Na primeira passada, fazemos `n-1` comparações; na segunda, `n-2`; na terceira `n-3`; e assim por diante. Logo, o tempo total gasto pelo algoritmo é proporcional a `(n-1) + (n-2) + ... + 2 + 1`. A soma desses termos é proporcional ao quadrado de `n`. Portanto, o desempenho computacional desse algoritmo varia de forma quadrática em relação ao tamanho do problema.

Em geral, usamos a notação "Big-O" para expressar como a complexidade de um algoritmo varia com o tamanho do problema. Assim, nesse caso em que o tempo computacional varia de forma quadrática com o tamanho do problema, dizemos que se trata de um algoritmo de ordem quadrática e expressamos isso escrevendo $O(n^2)$.

No melhor caso, quando o vetor fornecido estiver quase ordenado, o procedimento pode ser capaz de ordenar em uma única passada. Esse fato, no entanto, não pode ser usado para fazer uma análise de desempenho do algoritmo, pois o melhor caso representa uma situação muito particular.

Implementação recursiva

Ao analisar a forma como a ordenação bolha funciona, verificamos que o algoritmo procura resolver o problema da ordenação por partes. Inicialmente, o algoritmo coloca em sua posição correta (no final do vetor) o maior elemento, e o problema restante é semelhante ao inicial, só que com um vetor com menos elementos, formado pelos elementos `v[0],...,v[n-2]`.

Com base nessa observação, é fácil implementar um algoritmo de ordenação bolha recursivamente. Embora não seja a forma mais adequada de implementar

esse algoritmo, seu entendimento ajudará a compreender a idéia por trás do algoritmo de ordenação rápida que veremos mais adiante.

O algoritmo recursivo de ordenação bolha posiciona o elemento de maior valor e chama, recursivamente, o algoritmo para ordenar o vetor restante, com n-1 elementos.

```
/* Ordenação bolha recursiva */
void bolha_rec (int n, int* v)
{
  int j;
  int troca = 0;
   for (j=0; j<n-1; j++)
      if (v[j]>v[j+1]) {
         int temp = v[j];    /* troca */
         v[j] = v[j+1];
         v[j+1] = temp;
         troca = 1;
      }
  if (troca != 0)            /* houve troca */
     bolha_rec(n-1,v);
}
```

Algoritmo genérico

Esse mesmo algoritmo pode ser aplicado a vetores que guardam outras informações. O código escrito anteriormente pode ser reaproveitado, com exceção de alguns detalhes. Primeiro, a assinatura da função deve ser alterada, pois deixamos de ter um vetor de inteiros; segundo, a forma de comparação entre os elementos também deve ser alterada, pois não podemos, por exemplo, comparar duas cadeias de caracteres com o simples uso do operador relacional "maior que" (>).

Para aumentar o potencial de reutilização do nosso código, podemos reescrever o algoritmo de ordenação apresentado e torná-lo independente da informação armazenada no vetor. Vamos inicialmente discutir como podemos abstrair a função de comparação. O mesmo algoritmo para ordenação de inteiros apresentado pode ser reescrito com o uso de uma função auxiliar que faz a comparação. Em vez de comparar diretamente dois elementos com o operador "maior que", usamos uma função auxiliar que, dados dois elementos, verifica se o primeiro é maior do que o segundo.

```
/* Função auxiliar de comparação */
static int compara (int a, int b)
{
  if (a > b)
     return 1;
  else
     return 0;
}
```

```
/* Ordenação bolha (3a. versão) */
void bolha (int n, int* v)
{   int i, j;
  for (i=n-1; i>0; i--) {
      int troca = 0;
      for (j=0; j<i; j++)
         if (compara(v[j],v[j+1])) {
            int temp = v[j];      /* troca */
            v[j] = v[j+1];
            v[j+1] = temp;
            troca = 1;
         }
      if (troca == 0)            /* não houve troca */
         return;
  }
}
```

Dessa forma, já aumentamos o potencial de reutilização do algoritmo. Podemos, por exemplo, arrumar os elementos em ordem decrescente simplesmente reescrevendo a função compara. A idéia fundamental é escrever uma função de comparação que recebe dois elementos e verifica se há uma inversão de ordem entre o primeiro e o segundo. Assim, se tivéssemos um vetor de cadeia de caracteres para ordenar, poderíamos usar a seguinte função de comparação:

```
static int compara (char* a, char* b)
{
  if (strcmp(a,b) > 0)
     return 1;
  else
     return 0;
}
```

Consideremos agora um vetor de ponteiros para a estrutura Aluno:

```
struct aluno {
  char nome[81];
  char mat[8];
  char turma;
  char email[41];
};
typeset struct aluno Aluno;
```

Uma função de comparação, nesse caso, receberia como parâmetros dois ponteiros para a estrutura que representa um aluno e, segundo uma ordenação que usa o nome do aluno como chave de comparação, poderia ter a seguinte implementação:

```
static int compara (Aluno* a, Aluno* b)
{
  if (strcmp(a->nome,b->nome) > 0)
     return 1;
  else
     return 0;
}
```

Portanto, o uso de uma função auxiliar para realizar a comparação entre os elementos ajuda na obtenção de um código reutilizável. No entanto, só isso não é suficiente. Para o mesmo código poder ser aplicado a qualquer tipo de informação armazenada no vetor, precisamos tornar a implementação independente do tipo do elemento, isto é, precisamos tornar a própria função de ordenação (bolha) e a assinatura da função de comparação (compara) independentes do tipo do elemento.

Em C, a forma de generalizar o tipo é usar void*. Escreveremos o código de ordenação considerando que temos um ponteiro de qualquer tipo e passaremos para a função de comparação dois ponteiros genéricos, um para cada elemento que se deseja comparar. A função de ordenação, no entanto, precisa percorrer o vetor e, para tanto, precisamos passar para a função uma informação adicional – o tamanho, em número de bytes, de cada elemento. A assinatura da função de ordenação poderia então ser dada por:

```
void bolha (int n, void* v, int tam);
```

A função de ordenação, por sua vez, receberia dois ponteiros genéricos:

```
int compara (void* a, void* b);
```

Assim, se estamos ordenando vetores de inteiros, escrevemos a nossa função de comparação pela conversão do ponteiro genérico em um ponteiro de inteiro e pelo teste apropriado:

```
/* função de comparação para inteiros */
static int compara (void* a, void* b)
{
  int* p1 = (int*) a;
  int* p2 = (int*) b;
  if ((*p1) > (*p2))
     return 1;
  else
     return 0;
}
```

Se os elementos do vetor fossem ponteiros para a estrutura aluno, a função de comparação poderia ser:

```
/* função de comparação para ponteiros de alunos */
static int compara (void* a, void* b)
{
  Aluno** p1 = (Aluno**) a;
  Aluno** p2 = (Aluno**) b;
  if (strcmp((*p1)->nome,(*p2)->nome) > 0)
     return 1;
  else
     return 0;
}
```

Como dissemos, o código da função de ordenação necessita percorrer os elementos do vetor. O acesso a um determinado elemento i do vetor não pode mais ser feito diretamente por v[i]. Dado o endereço do primeiro elemento do vetor, devemos incrementar esse endereço de i*tam bytes para ter o endereço do elemento i. Podemos então escrever uma função auxiliar que faz esse incremento de endereço. Essa função recebe como parâmetros o endereço inicial do vetor, o índice do elemento cujo endereço se quer alcançar e o tamanho (em bytes) de cada elemento. A função retorna o endereço do elemento especificado. Uma parte sutil, porém necessária, dessa função é que, para incrementar o endereço genérico de um determinado número de bytes, precisamos antes, temporariamente, converter esse ponteiro em ponteiro para caractere (pois um caractere ocupa um byte). O código dessa função auxiliar pode ser dado por:

```
static void* acessa (void* v, int i, int tam)
{
  char* t = (char*)v;
  t += tam*i;
  return (void*)t;
}
```

A função de ordenação identifica a ocorrência de inversões entre elementos e realiza uma troca entre os valores. O código que realiza a troca também tem de ser pensado de forma genérica, pois, como não sabemos o tipo de cada elemento, não temos como declarar a variável temporária para poder realizar a troca. Uma alternativa é fazer a troca dos valores byte a byte (ou caractere a caractere). Para tanto, podemos definir uma outra função auxiliar que recebe os ponteiros genéricos dos dois elementos que devem ter seus valores trocados, além do tamanho de cada um.

```
static void troca (void* a, void* b, int tam)
{
  char* v1 = (char*) a;
  char* v2 = (char*) b;
  int i;
  for (i=0; i<tam; i++) {
     char temp = v1[i];
     v1[i] = v2[i];
     v2[i] = temp;
  }
}
```

Assim, podemos escrever o código da nossa função de ordenação genérica. Falta, no entanto, um último detalhe. As funções auxiliares `acessa` e `troca` são realmente genéricas e independem da informação efetivamente armazenada no vetor. Entretanto, a função de comparação deve ser especializada para cada tipo de informação, conforme ilustrado. A assinatura dessa função é genérica, mas a sua implementação deve, naturalmente, levar em conta a informação armazenada para que a comparação tenha sentido. Portanto, para generalizar a implementação da função de ordenação, não podemos chamar uma função de comparação específica. A solução é passar, via parâmetro, qual função *callback* de comparação deve ser chamada. A função de comparação tem a assinatura:

```
int compara (void*, void*);
```

Com isso, a assinatura da função genérica de ordenação, recebendo a *callback* como parâmetro, passa a ser:

```
void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*))
```

onde `cmp` representa a variável do tipo ponteiro para a função de comparação.

Agora, sim, podemos escrever nossa função de ordenação genérica:

```
/* Ordenação bolha (genérica) */
void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*))
{  int i, j;
  for (i=n-1; i>0; i--) {
      int fez_troca = 0;
      for (j=0; j<i; j++) {
         void* p1 = acessa(v,j,tam);
         void* p2 = acessa(v,j+1,tam);
         if (cmp(p1,p2)) {
           troca(p1,p2,tam);
           fez_troca = 1;
         }
      }
      if (fez_troca == 0)   /* não houve troca */
         return;
  }
}
```

Esse código genérico pode ser usado para ordenar vetores com qualquer informação. Para exemplificar, vamos usá-lo para ordenar um vetor de números reais. Para isso, temos de escrever o código da função que faz a comparação, agora especializada para números reais:

```
static int compara_reais (void* a, void* b)
{
  float* p1 = (float*) a;
  float* p2 = (float*) b;
  if ((*p1) > (*p2))
     return 1;
  else
     return 0;
}
```

Podemos, então, chamar a função para ordenar um vetor v de n números reais:

```
  ...
  bolha_gen(n,v,sizeof(float),compara_reais);
  ...
```

Ordenação rápida

Assim como o algoritmo anterior, o algoritmo "ordenação rápida", "*quick sort*", que iremos discutir agora, procura resolver o problema da ordenação por partes. No entanto, enquanto o algoritmo de ordenação bolha coloca em sua posição (no final do vetor) o maior elemento, a ordenação rápida faz isso com um elemento arbitrário x, chamado de pivô. Por exemplo, podemos escolher como pivô o primeiro elemento do vetor e posicionar esse elemento em sua posição correta em uma primeira passada.

Suponha que esse elemento, x, deva ocupar a posição i do vetor, de acordo com a ordenação, ou seja, que essa seja a sua posição definitiva. Sem ordenar o vetor completamente, esse fato pode ser reconhecido quando todos os elementos v[0], ... v[i-1] são menores do que x, e todos os elementos v[i+1], ..., v[n-1] são maiores do que x. Caso se suponha que x já esteja na sua posição correta, com índice i, há dois problemas menores para serem resolvidos: ordenar os (sub)vetores formados por v[0], ... v[i-1] e por v[i+1], ..., v[n-1]. Esses subproblemas são resolvidos (recursivamente) de forma semelhante, cada vez com vetores menores, e o processo continua até os vetores que devem ser ordenados terem zero ou um elemento, caso no qual sua ordenação já está concluída.

A grande vantagem desse algoritmo é que ele pode ser muito eficiente. O melhor caso ocorre quando o elemento pivô representa o valor mediano do conjunto dos elementos do vetor. Se isso acontece, após o posicionamento do pivô em sua posição, restarão dois subvetores para serem ordenados, ambos com o núme-

ro de elementos reduzido à metade, em relação ao vetor original. Pode-se mostrar que, nesse melhor caso, o esforço computacional do algoritmo é proporcional a *n log(n)*, e dizemos que o algoritmo é *O(n log(n))* – um desempenho muito superior ao $O(n^2)$ apresentado pelo algoritmo de ordenação bolha. Infelizmente, não temos como garantir que o pivô seja o mediano. No pior caso, o pivô pode sempre ser, por exemplo, o maior elemento, e recaímos no algoritmo de ordenação bolha. No entanto, é possível mostrar que o algoritmo *quicksort* ainda apresenta, no caso médio, um desempenho *O(n log(n))*.

A versão do *quick sort* que vamos apresentar aqui usa `x=v[0]` como o primeiro elemento a ser colocado em sua posição correta. O processo compara os elementos `v[1]`, `v[2]`, ... até encontrar um elemento `v[a]>x`. Então, a partir do final do vetor, compara os elementos `v[n-1]`, `v[n-2]`, ... até encontrar um elemento `v[b]<=x`. Nesse ponto, `v[a]` e `v[b]` são trocados e a busca continua, para cima a partir de `v[a+1]` e para baixo a partir de `v[b-1]`. Em algum momento, a busca termina, porque os pontos de busca se encontrarão (`b<a`). Nesse momento, a posição correta de `x` está definida, e os valores `v[0]` e `v[b]` são trocados.

Vamos usar o mesmo exemplo da seção anterior:

```
(0-7) 25 48 37 12 57 86 33 92
```

onde indicamos com `(0-7)` que se trata do vetor inteiro, de `v[0]` a `v[7]`. Podemos começar a executar o algoritmo com vistas a determinar a posição correta de `x=v[0]=25`. Partindo do início do vetor, já temos, na primeira comparação, `48>25` (`a=1`). Partindo do final do vetor, na direção oposta, temos `25<92`, `25<33`, `25<86`, `25<57` e finalmente, `12<=25 (b=3)`.

```
(0-7) 25 48 37 12 57 86 33 92
         a↑    b↑
```

Trocamos então `v[a]=48` e `v[b]=12`, incrementando `a` em uma unidade e decrementando `b` de uma unidade. Os elementos do vetor ficam com a seguinte disposição:

```
(0-7) 25 12 37 48 57 86 33 92
           a,b↑
```

Na continuação, temos `37>25 (a=2)`. Pelo outro lado, chegamos também a `37` e temos `37>25` e `12<=25`. Nesse ponto, verificamos que os índices `a` e `b` se cruzaram, agora com `b<a`.

```
(0-7) 25 12 37 48 57 86 33 92
         b↑ a↑
```

Assim, todos os elementos de 37 (inclusive) em diante são maiores do que 25, e todos os elementos de 12 (inclusive) para trás são menores do que 25 – com exceção do próprio 25, é claro. A próxima etapa troca o pivô, v[0]=25, com o último dos valores menores do que 25 encontrado: v[b]=12. Temos:

```
(0-7)12 25 37 48 57 86 33 92
```

com 25 em sua posição correta e dois vetores menores para ordenar. Valores menores do que 25:

```
(0-0) 12
```

E valores maiores:

```
(2-7) 37 48 57 86 33 92
```

Nesse caso, em particular, o primeiro vetor (com apenas um elemento: (0-0)) já se encontra ordenado. O segundo vetor (2-7) pode ser ordenado de forma semelhante:

```
(2-7) 37 48 57 86 33 92
```

Devemos achar a posição correta de 37. Para isso, identificamos o primeiro elemento maior do que 37, ou seja, 48, e o último menor do que 37, ou seja, 33.

```
(2-7) 37 48 57 86 33 92
         a↑          b↑
```

Trocamos os elementos e atualizamos os índices:

```
(2-7) 37 33 57 86 48 92
            a↑ b↑
```

Ao continuar o processo, verificamos que 37<57 e 37<86, 37<57, mas 37>=33. Identificamos novamente que a e b se cruzaram.

```
(2-7) 37 33 57 86 48 92
         b↑ a↑
```

Assim, a posição correta de 37 é a posição ocupada por v[b], e os dois elementos devem ser trocados:

```
(2-7) 33 37 57 86 48 92
```

restam os vetores

```
(2-2) 33
```

e

```
(4-7) 57 86 48 92
```

para serem ordenados.

O processo continua até que o vetor original esteja totalmente ordenado.

```
(0-7) 12 25 33 37 48 57 86 92
```

A implementação do *quick sort* é normalmente recursiva, para facilitar a ordenação dos dois vetores menores encontrados. A seguir, apresentamos uma possível implementação do algoritmo, com a adoção do primeiro elemento como pivô.

```
/* Ordenação rápida */
void rapida (int n, int* v)
{
  if (n <= 1)
     return;
  else {
     int x = v[0];
     int a = 1;
     int b = n-1;
     do {
        while (a < n && v[a] <= x) a++;
        while (v[b]> x) b--;
        if (a < b) {           /* faz troca */
           int temp = v[a];
           v[a] = v[b];
           v[b]= temp;
           a++; b--;
        }
     } while (a <= b);

     /* troca pivô */
     v[0] = v[b];
     v[b]= x;

     /* ordena subvetores restantes */
     rapida(b,v);
     rapida(n-a,&v[a]);
  }
}
```

Devemos observar que, para deslocar o índice a para a direita, fizemos o teste:

```
while (a < n && v[a] <= x)
```

enquanto, para deslocar o índice b para a esquerda, fizemos apenas:

```
while (v[b]> x)
```

O teste adicional no deslocamento para a direita é necessário porque o pivô pode ser o elemento de maior valor, nunca ocorrendo a situação v[a]<=x, o que nos faria acessar posições além dos limites do vetor. No deslocamento para a esquerda, um teste adicional tipo b>=0 não é necessário, pois, na nossa implementação, v[0] é o pivô, e isso impede que b assuma valores negativos (teremos, pelo menos, v[0]==x).

Algoritmo genérico da biblioteca padrão

O *quick sort* é o algoritmo de ordenação mais utilizado no desenvolvimento de aplicações. Mesmo quando temos os dados organizados em listas encadeadas e precisamos colocá-los de forma ordenada, em geral, optamos por criar um vetor temporário com ponteiros para os nós da lista, fazer a ordenação com o *quick sort* e reencadear os nós montando a lista ordenada.

Devido à sua grande utilidade, a biblioteca padrão de C disponibiliza, via a interface *stdlib.h*, uma função que ordena vetores por meio desse algoritmo. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os princípios discutidos na implementação do algoritmo de ordenação bolha genérico que discutimos na seção anterior. O protótipo da função disponibilizada pela biblioteca é[1]:

```
void qsort (void *v, int n, int tam,
            int (*cmp)(const void*, const void*)
           );
```

Os parâmetros de entrada dessa função são:

- v: ponteiro para o primeiro elemento do vetor que se deseja ordenar. Como não se sabe, *a priori*, o tipo dos elementos do vetor, temos um ponteiro genérico – void*.
- n: número de elementos do vetor.
- tam: tamanho, em bytes, de cada elemento do vetor.
- cmp: ponteiro para a função responsável por comparar dois elementos do vetor. Em C, o nome de uma função representa o ponteiro da função. Esse ponteiro pode ser armazenado em uma variável, possibilitando chamar a função indiretamente. Como era de se esperar, a biblioteca não sabe compa-

[1] A rigor, os parâmetros *n* e tam são do tipo size_t.

rar dois elementos do vetor (ela desconhece o tipo desses elementos). Fica a cargo do cliente da função de ordenação escrever a função de comparação, que tem de ter o seguinte protótipo:

```
int nome (const void*, const void*);
```

O parâmetro cmp recebido pela função qsort é um ponteiro para uma função com esse protótipo. Assim, para usar a função de ordenação da biblioteca, temos de escrever uma função para receber dois ponteiros genéricos, void*, os quais representam ponteiros para os dois elementos que se deseja comparar. O modificador de tipo const aparece no protótipo apenas para garantir que essa função não modificará os valores dos elementos (devem ser tratados como valores constantes). Essa função deve ter como valor de retorno < 0, 0, ou > 0, dependendo de se o primeiro elemento for menor, igual, ou maior do que o segundo, respectivamente, de acordo com o critério de ordenação adotado.

Para ilustrar a utilização da função qsort, vamos considerar alguns exemplos. O código a seguir ilustra a utilização da função para ordenar valores reais. Nesse caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para float.

```
/* Ilustra uso do algoritmo qsort */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de reais */
static int comp_reais (const void* p1, const void* p2)
{
  /* converte ponteiros genéricos para ponteiros de float */
  float *f1 = (float*)p1;
  float *f2 = (float*)p2;
  /* dados os ponteiros de float, faz a comparação */
  if (*f1 < *f2) return -1;
  else if (*f1 > *f2) return 1;
  else return 0;
}

/* programa que faz a ordenação de um vetor */
int main (void)
{
  int i;
  float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};
  qsort(v,8,sizeof(float),comp_reais);

  printf("Vetor ordenado: ");
  for (i=0; i<8; i++)
     printf("%g ",v[i]);
  printf("\n");
  return 0;
}
```

Vamos agora considerar que temos um vetor de alunos e desejamos ordená-lo usando o nome do aluno como chave de comparação. A estrutura que representa um aluno pode ser dada por:

```
struct aluno {
  char nome[81];
  char mat[8];
  char turma;
  char email[41];
};
typedef struct aluno Aluno;
```

Vamos analisar duas situações. Na primeira, consideraremos a existência de um vetor da estrutura (por exemplo, `Aluno vet[N];`). Nesse caso, cada elemento do vetor é do tipo `Aluno`, e os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `Aluno`. Essa função de comparação pode ser dada por:

```
/* Função de comparação: elemento é do tipo Aluno */
static int comp_alunos (const void* p1, const void* p2)
{
  /* converte ponteiros genéricos para ponteiros de Aluno */
  Aluno *a1 = (Aluno*)p1;
  Aluno *a2 = (Aluno*)p2;
  /* dados os ponteiros de Aluno, faz a comparação */
  return strcmp(a1->nome,a2->nome);
}
```

Em uma segunda situação, podemos considerar que temos um vetor de ponteiros para a estrutura `aluno` (por exemplo, `Aluno* vet[N];`). Agora, cada elemento do vetor é um ponteiro para o tipo `Aluno`, e a função de comparação tem de tratar uma indireção a mais. Aqui, os dois ponteiros genéricos passados para a função de comparação representam ponteiros de ponteiros para `Aluno`.

```
/* Função de comparação: elemento é do tipo Aluno* */
static int comp_alunos (const void* p1, const void* p2)
{
  /* converte p/ ponteiros de ponteiros de Aluno */
  Aluno **a1 = (Aluno**)p1;
  Aluno **a2 = (Aluno**)p2;
  /* dados os ponteiros de ponteiro de Aluno, faz a comparação */
  return strcmp((*a1)->nome,(*a2)->nome);
}
```

17

Busca

Neste capítulo, discutiremos diferentes estratégias para efetuar a busca de um elemento em um determinado conjunto de dados. A operação de busca é encontrada com muita freqüência em aplicações computacionais, sendo, portanto, importante estudar estratégias distintas para efetuá-la. Por exemplo, um programa de controle de estoque pode buscar, dado um código numérico ou um nome, a descrição e as características de um determinado produto. Se tivermos um grande número de produtos cadastrados, o método para efetuar a busca deverá ser eficiente; caso contrário a busca poderá ser muito demorada e inviabilizar, assim, a operação.

Inicialmente, consideraremos ter nossos dados armazenados em um vetor e discutiremos os algoritmos de busca que podemos utilizar. A seguir, discutiremos a utilização de árvores binárias de busca, que são estruturas de árvores projetadas para dar suporte a operações de busca de forma eficiente. No próximo capítulo, discutiremos as estruturas conhecidas como tabelas de dispersão (*hash*), que podem, como veremos, realizar buscas de forma extremamente eficiente.

Busca em vetor

Nesta seção, apresentaremos os algoritmos de busca em vetor. Dado um vetor `vet` com `n` elementos, desejamos saber se um determinado elemento `elem` está ou não presente no vetor. Se estiver, a função de busca retorna em que posição no vetor o elemento se encontra.

Busca linear

A forma mais simples de fazer uma busca em um vetor consiste em percorrer o vetor, elemento a elemento, para verificar se o elemento de interesse é igual a um

dos elementos do vetor. Esse algoritmo pode ser implementado conforme ilustrado pelo código a seguir, com a consideração de um vetor de números inteiros. A função apresentada tem como valor de retorno o índice do vetor no qual foi encontrado o elemento; se o elemento não for encontrado, o valor de retorno é –1.

```
int busca (int n, int* vet, int elem)
{
  int i;

  for (i=0; i<n; i++) {
     if (elem == vet[i])
        return i;        /* elemento encontrado */
  }

  /* percorreu todo o vetor e não encontrou elemento */
  return -1;
}
```

Esse algoritmo de busca é extremamente simples, mas será muito ineficiente quando o número de elementos no vetor for muito grande. Isso porque o algoritmo (a função, no caso) pode ter de percorrer todos os elementos do vetor para verificar se um determinado elemento está ou não presente. No pior caso, será necessário realizar *n* comparações, em que *n* representa o número de elementos no vetor. Portanto, o desempenho computacional desse algoritmo varia linearmente em relação ao tamanho do problema. Chamamos esse algoritmo de busca linear, e sua complexidade é expressa por $O(n)$.

Além do pior caso, podemos analisar o caso médio, isto é, o caso que ocorre na média. Já vimos que o algoritmo em questão requer *n* comparações quando o elemento não está presente no vetor. No caso de o elemento estar presente, quantas operações de comparação são, em média, necessárias? Na média, podemos concluir que são necessárias $n/2$ comparações. Em termos de ordem de complexidade, no entanto, continuamos a ter uma variação linear, isto é, $O(n)$, pois dizemos que $O(k\,n)$, onde *k* é uma constante relativamente pequena, é igual a $O(n)$.

Em diversas aplicações reais, precisamos de algoritmos de busca mais eficientes. Seria possível melhorar a eficiência do algoritmo de busca mostrado? Infelizmente, se os elementos estiverem armazenados em uma ordem aleatória no vetor, não temos como melhorar o algoritmo de busca, pois precisamos verificar todos os elementos. No entanto, se assumirmos, por exemplo, o armazenamento dos elementos em ordem crescente, podemos concluir que um elemento não está presente no vetor se acharmos um elemento maior, pois, se o elemento buscado estivesse presente, ele precederia um elemento maior na ordem do vetor.

O código a seguir ilustra a implementação da busca linear a partir da suposição de que os elementos do vetor estão ordenados (vamos assumir ordem crescente).

```
int busca_ord (int n, int* vet, int elem)
{
  int i;

  for (i=0; i<n; i++) {
     if (elem == vet[i])
        return i;               /* elemento encontrado */
     else if (elem < vet[i])
        return -1;              /* interrompe busca */
  }

  /* percorreu todo o vetor e não encontrou elemento */
  return -1;
}
```

No caso de o elemento procurado não pertencer ao vetor, esse segundo algoritmo apresenta um desempenho ligeiramente superior ao primeiro, mas a ordem dessa versão do algoritmo continua sendo linear – $O(n)$. No entanto, se os elementos do vetor estão ordenados, existe um algoritmo muito mais eficiente, que será apresentado a seguir.

Busca binária

Se os elementos do vetor estiverem ordenados, podemos aplicar um algoritmo mais eficiente para realizar a busca. Trata-se do algoritmo de busca binária. A idéia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor. Se o elemento que buscamos for menor que o elemento do meio, sabemos que, se o elemento estiver presente no vetor, ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor. Se concluirmos que o elemento está em uma das partes do vetor, repetimos o procedimento considerando apenas a parte restante: comparamos o elemento buscado com o elemento armazenado no meio dessa parte. Esse procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrar o elemento ou chegar a uma parte do vetor com tamanho zero.

O código a seguir ilustra uma implementação de busca binária em um vetor de valores inteiros ordenados de forma crescente.

```
int busca_bin (int n, int* vet, int elem)
{
  /* no início consideramos todo o vetor */
  int ini = 0;
  int fim = n-1;
  int meio;
```

```
  /* enquanto a parte restante for maior que zero */
  while (ini <= fim) {
     meio = (ini + fim) / 2;
     if (elem < vet[meio])
        fim = meio - 1;          /* ajusta posição final */
     else if (elem > vet[meio])
        ini = meio + 1;          /* ajusta posição inicial */
     else
        return meio;             /* elemento encontrado */
  }
  /* não encontrou: restou parte de tamanho zero */
  return -1;
}
```

O desempenho desse algoritmo é muito superior ao de busca linear. Novamente, o pior caso caracteriza-se pela situação de o elemento que buscamos não estar no vetor. Quantas vezes precisamos repetir o procedimento de subdivisão para concluir que o elemento não está presente no vetor? A cada repetição, a parte considerada na busca é dividida pela metade. A tabela a seguir mostra o tamanho do vetor a cada repetição do laço do algoritmo.

Repetição	Tamanho do problema
1	*n*
2	*n/2*
3	*n/4*
...	...
log n	*1*

Assim, são necessárias *log n* repetições. Como fazemos um número constante de comparações a cada ciclo (duas comparações por ciclo), podemos concluir que a ordem desse algoritmo é *O(log n)*.

O algoritmo de busca binária consiste em repetir o mesmo procedimento recursivamente, o qual pode ser implementado de forma recursiva. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse algoritmo, a implementação recursiva é mais sucinta e vale a pena ser apresentada. Na implementação recursiva, temos dois casos a serem tratados. No primeiro, a busca deve continuar na primeira metade do vetor, logo chamamos a função recursivamente passando como parâmetros o número de elementos dessa primeira parte restante e o mesmo ponteiro para o primeiro elemento, pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo. No segundo caso, a busca deve continuar apenas na segunda parte do vetor, logo passamos na chamada recursiva, além do número de elementos restantes, um ponteiro para o

primeiro elemento dessa segunda parte. Para simplificar, uma primeira versão apenas informa se o elemento pertence ou não ao vetor, e tem como valor de retorno falso (0) ou verdadeiro (1). Uma possível implementação que usa essa estratégia é mostrada a seguir.

```
int pertence_rec (int n, int* vet, int elem)
{
  /* testa condição de contorno: parte com tamanho zero */
  if (n <= 0)
     return 0;
  else {
     /* deve buscar o elemento do meio */
     int meio = n / 2;
     if (elem < vet[meio])
        return pertence_rec (meio,vet,elem);
     else if (elem > vet[meio])
        return pertence_rec (n-1-meio, &vet[meio+1],elem);
     else
        return 1;             /* elemento encontrado */
  }
}
```

Em particular, devemos notar a expressão &vet[meio+1] que, como sabemos, resulta em um ponteiro para o primeiro elemento da segunda parte do vetor.

Se quisermos que a função tenha como valor de retorno o índice do elemento, devemos acertar o valor retornado pela chamada recursiva na segunda parte do vetor. Uma implementação dessa função de busca é apresentada a seguir:

```
int busca_bin_rec (int n, int* vet, int elem)
{
  /* testa condição de contorno: parte com tamanho zero */
  if (n <= 0)
     return -1;
  else {
     /* deve buscar o elemento do meio */
     int meio = n / 2;
     if (elem < vet[meio])
        return busca_bin_rec(meio,vet,elem);
     else if (elem > vet[meio])
     {
        int r = busca_bin_rec(n-1-meio, &vet[meio+1],elem);
        if (r<0) return -1;
        else     return meio+1+r;
     }
     else
        return meio;              /* elemento encontrado */
  }
}
```

Devemos finalmente salientar que, se tivermos os dados armazenados em uma lista encadeada, só temos a alternativa de implementar um algoritmo de busca linear, mesmo se os elementos estiverem ordenados. Portanto, a lista encadeada não é uma boa opção para estruturar nossos dados, se desejarmos realizar muitas operações de busca. A estrutura dinâmica apropriada para a realização de busca é a árvore binária de busca, que será discutida mais adiante. Antes, porém, vamos descrever o algoritmo genérico para busca binária em vetor.

Algoritmo genérico

A biblioteca padrão de C disponibiliza, via a interface *stdlib.h*, uma função que faz a busca binária de um elemento em um vetor. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os mesmos princípios discutidos no capítulo anterior. O protótipo da função de busca binária da biblioteca é[1]:

```
void* bsearch (void* info, void *v, int n, int tam,
               int (*cmp)(const void*, const void*)
               );
```

Se o elemento for encontrado no vetor, a função tem como valor de retorno o endereço do elemento no vetor; caso o elemento não seja encontrado, o valor de retorno é `NULL`. De modo análogo à função `qsort`, apresentada no capítulo anterior, os parâmetros de entrada dessa função são:

- `info`: ponteiro para a informação que se deseja buscar no vetor – representa a chave de busca;
- `v`: ponteiro para o primeiro elemento do vetor no qual a busca será feita. Os elementos do vetor têm de estar ordenados, segundo o critério de ordenação adotado pela função de comparação descrita a seguir;
- `n`: número de elementos do vetor;
- `tam`: tamanho, em bytes, de cada elemento do vetor;
- `cmp`: ponteiro para a função responsável por comparar a informação que se busca com um elemento do vetor. O primeiro parâmetro dessa função é sempre o endereço da informação que se busca, e o segundo é um ponteiro para um dos elementos do vetor. O critério de comparação adotado por essa função deve ser compatível com o critério de ordenação do vetor. Essa

[1] A rigor, os parâmetros `info` e `v` têm modificadores `const`, e os parâmetros `n` e `tam` são do tipo `size_t`.

função deve ter como valor de retorno < 0, 0 ou > 0, dependendo de se a informação que se busca for menor, igual ou maior que a informação armazenada no elemento, respectivamente.

Para ilustrar a utilização da função bsearch, vamos inicialmente considerar um vetor de valores inteiros em ordem crescente. Nesse caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para int, e a função de comparação recebe dois ponteiros para int.

```
/* Ilustra uso do algoritmo bsearch */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de inteiros */
static int comp_int (const void* p1, const void* p2)
{
  /* converte ponteiros genéricos para ponteiros de int */
  int *info = (int*)p1;
  int *elem = (int*)p2;
  /* dados os ponteiros de int, faz a comparação */
  if (*info < *elem) return -1;
  else if (*info > *elem) return 1;
  else return 0;
}

/* programa que faz a busca em um vetor */
int main (void)
{
  int v[8] = {12,25,33,37,48,57,86,92};
  int e = 57;     /* informação que se deseja buscar */
  int* p;

  p = (int*)bsearch(&e,v,8,sizeof(int),comp_int);

  if (p == NULL)
     printf("Elemento não encontrado.\n");
  else
     printf("Elemento encontrado no índice: %d\n", p-v);
  return 0;
}
```

Devemos notar que o índice do elemento, se encontrado no vetor, pode ser extraído ao subtrair o ponteiro do elemento do ponteiro do primeiro elemento (p-v). Essa aritmética de ponteiros é válida aqui porque podemos garantir que ambos os ponteiros armazenam endereços de memória de um mesmo vetor. A di-

ferença entre os ponteiros representa a "distância" em que os elementos estão armazenados na memória.

Vamos agora considerar uma busca em um vetor de ponteiros para alunos. A estrutura que representa um aluno pode ser dada por:

```
struct aluno {
  char nome[81];
  char mat[8];
  char turma;
  char email[41];
};
typedef struct aluno Aluno;
```

Como o vetor está ordenado segundo os nomes dos alunos, podemos buscar a ocorrência de um determinado aluno passando para a função de busca um nome e o vetor. A função de comparação então receberá dois ponteiros que referenciam tipos distintos: um ponteiro para uma cadeia de caracteres e um ponteiro para um elemento do vetor (no caso será um ponteiro para ponteiro de aluno, ou seja, um `Aluno**`).

```
/* Função de comparação: char* e Aluno** */
static int comp_alunos (const void* p1, const void* p2)
  /* converte ponteiros genéricos para ponteiros específicos */
  char* s = (char*)p1;
  Aluno **pa = (Aluno**)p2;
  /* faz a comparação */
  return strcmp(s,(*pa)->nome);
}
```

Conforme observamos, o tipo de informação a ser buscada nem sempre é igual ao tipo do elemento; para dados complexos, em geral não é. A informação buscada geralmente representa um campo da estrutura armazenada no vetor (ou da estrutura apontada por elementos do vetor).

Árvore binária de busca

Como vimos, o algoritmo de busca binária apresentado na seção anterior tem bom desempenho computacional e deve ser usado quando temos os dados ordenados armazenados em um vetor. Contudo, se precisarmos inserir e remover elementos da estrutura e ao mesmo tempo dar suporte a funções de busca eficientes, a estrutura de vetor (e, conseqüentemente, o uso do algoritmo de busca binária) não se mostra adequada. Para inserir um novo elemento em um vetor ordenado, temos de rearrumar os elementos no vetor para abrir espaço para a inserção do

novo elemento. Uma situação análoga ocorre quando removemos um elemento do vetor. Precisamos portanto de uma estrutura dinâmica que dê suporte a operações de busca.

Um dos resultados que apresentamos anteriormente foi o da relação entre o número de nós de uma árvore binária e sua altura. A cada nível, o número (potencial) de nós vai dobrando, de maneira que uma árvore binária de altura *h* pode ter um número de nós dado por:

$$1 + 2 + 2^2 + \ldots + 2^{h-1} + 2^h = 2^{h+1}-1$$

Assim, dizemos que uma árvore binária de altura *h* pode ter no máximo $O(2^h)$ nós, ou, por outro lado, que uma árvore binária com *n* nós pode ter uma altura mínima de $O(\log n)$. Essa relação entre o número de nós e a altura mínima da árvore é importante porque, se as condições forem favoráveis, podemos alcançar qualquer um dos *n* nós de uma árvore binária a partir da raiz em, no máximo, $O(\log n)$ passos. Se tivéssemos os *n* nós em uma lista linear, o número máximo de passos seria $O(n)$ e, para os valores de *n* encontrados na prática, $\log n$ é muito menor do que *n*.

A altura de uma árvore é, certamente, uma medida do tempo necessário para encontrar um dado nó. No entanto, é importante observar que para acessar qualquer nó de maneira eficiente é necessário ter árvores binárias "balanceadas", com o número de nós à esquerda igual, ou próximo ao número de nós à direita (inclusive para as subárvores, recursivamente). Lembramos que o número mínimo de nós de uma árvore binária de altura *h* é $h+1$, e assim a altura máxima de uma árvore com *n* nós é dada por $O(n)$. Esse caso extremo corresponde à árvore "degenerada", em que todos os nós têm apenas *1* filho, com exceção da (única) folha.

As árvores binárias consideradas nesta seção têm uma propriedade fundamental: o valor associado à raiz é sempre maior do que o valor associado a qualquer nó da subárvore à esquerda (*sae*) e é sempre menor do que o valor associado a qualquer nó da subárvore à direita (*sad*). Essa propriedade garante que, quando a árvore é percorrida em ordem simétrica (*sae* - *raiz* - *sad*), os valores são encontrados em ordem crescente.

Uma variação possível permite a repetição de valores na árvore: o valor associado à raiz é sempre maior do que o valor associado a qualquer nó da *sae* e é sempre menor ou igual ao valor associado a qualquer nó da *sad*. Nesse caso, como a repetição de valores é permitida, quando a árvore é percorrida em ordem simétrica, os valores são encontrados em ordem não decrescente.

Ao usar essa propriedade de ordem, a busca de um valor em uma árvore pode ser feita de forma eficiente. Para procurar um valor numa árvore, comparamos o valor que buscamos ao valor associado à raiz. Em caso de igualdade, o valor foi encontrado; se o valor dado for menor que o valor associado à raiz, a busca continua na *sae*; caso contrário, se o valor associado à raiz for menor, a busca continua

na *sad*. Por essa razão, essas árvores são freqüentemente chamadas de árvores binárias de busca.

Naturalmente, a ordem a que fizemos referência anteriormente é dependente da aplicação. Se a informação a ser armazenada em cada nó da árvore for um número inteiro, podemos usar o habitual operador relacional *menor que* ("<"). Porém, se tivermos de considerar casos em que a informação é mais complexa, uma função de comparação específica deve ser empregada.

Operações em árvores binárias de busca

Para exemplificar a implementação de operações em árvores binárias de busca, vamos considerar o caso em que a informação associada a um nó é um número inteiro e não vamos considerar a possibilidade de repetição de valores associados aos nós da árvore. A Figura 17.1 ilustra uma árvore de busca de valores inteiros.

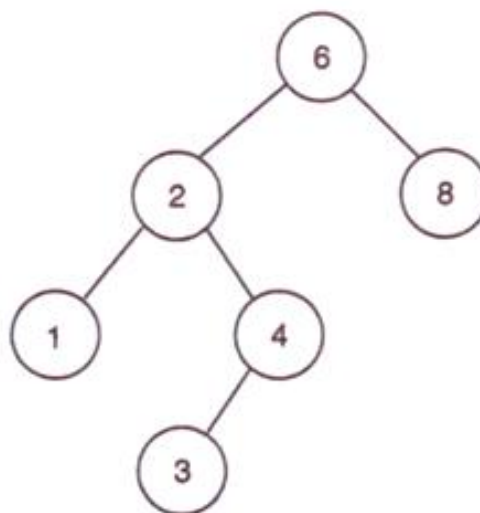



Figura 17.1 *Exemplo de árvore binária de busca.*

O tipo da árvore binária pode então ser dado por:

```
struct arv {
  int info;
  struct arv* esq;
  struct arv* dir;
};

typedef struct arv Arv;
```

A árvore é representada pelo ponteiro para o nó raiz. A árvore vazia é inicializada pela atribuição de `NULL` à variável que representa a árvore. Uma função simples para criar uma árvore vazia é mostrada a seguir:

```
Arv* abb_cria (void)
{
  return NULL;
}
```

Caso se suponha a existência de uma árvore binária de busca já construída, podemos imprimir os valores da árvore em ordem crescente percorrendo os nós em ordem simétrica:

```
void abb_imprime (Arv* a)
{
  if (a != NULL) {
     abb_imprime(a->esq);
     printf("%d\n",a->info);
     abb_imprime(a->dir);
  }
}
```

Essas são funções análogas às vistas para árvores binárias comuns, pois não exploram a propriedade de ordenação das árvores de busca. Todavia, as operações que nos interessam analisar em detalhes são:

- `busca`: função que busca um elemento na árvore;
- `insere`: função que insere um novo elemento na árvore;
- `retira`: função que retira um elemento da árvore.

Operação de busca

A operação para buscar um elemento na árvore explora a propriedade de ordenação da árvore, com um desempenho computacional proporcional à sua altura (*O(log n)* para o caso de árvore balanceada). Uma implementação da função de busca é dada por:

```
Arv* abb_busca (Arv* r, int v)
{
  if (r == NULL) return NULL;
  else if (r->info > v) return abb_busca (r->esq, v);
  else if (r->info < v) return abb_busca (r->dir, v);
  else return r;
}
```

Operação de inserção

A operação de inserção adiciona um elemento na árvore na posição correta para que a propriedade fundamental seja mantida. Para inserir um valor `v` em uma ár-

vore, usamos sua estrutura recursiva e a ordenação especificada na propriedade fundamental. Se a (sub)árvore for vazia, deve ser substituída por uma árvore cujo único nó (o nó raiz) contém o valor v. Se a árvore não for vazia, comparamos v ao valor na raiz da árvore e inserimos v na *sae* ou na *sad*, conforme o resultado da comparação. A função a seguir ilustra a implementação dessa operação. A função tem como valor de retorno o eventual novo nó raiz da (sub)árvore.

```
Arv* abb_insere (Arv* a, int v)
{
  if (a==NULL) {
     a = (Arv*)malloc(sizeof(Arv));
     a->info = v;
     a->esq = a->dir = NULL;
  }
  else if (v < a->info)
     a->esq = abb_insere(a->esq,v);
  else  /* v < a->info */
     a->dir = abb_insere(a->dir,v);
  return a;
}
```

Mais uma vez, salientamos a necessidade de atualizar os ponteiros para as subárvores à esquerda ou à direita quando da chamada recursiva da função, pois a função de inserção pode alterar o valor do ponteiro para a raiz da (sub)árvore.

Operação de remoção

Outra operação a ser analisada é a que permite retirar um determinado elemento da árvore. Essa operação também deve ter como valor de retorno a eventual nova raiz da árvore, mas sua implementação é mais complexa que a inserção. De novo, devemos pensar essa implementação com base na definição recursiva da árvore. Se a árvore for vazia, nada tem de ser feito, pois o elemento não está presente na árvore. Se a árvore não for vazia, comparamos o valor armazenado no nó raiz ao valor que se deseja retirar da árvore. Se o valor associado à raiz for maior do que o valor a ser retirado, chamamos a função recursivamente para retirar o elemento da subárvore à esquerda. Se o valor da raiz for menor, retiramos o elemento da subárvore à direita. Finalmente, se o valor associado à raiz for igual, encontramos o elemento a ser retirado e devemos efetuar essa operação. Portanto, estaremos sempre retirando um nó raiz de uma (sub)árvore.

Nesse caso, existem três situações possíveis. A primeira, e mais simples, é quando se deseja retirar uma raiz que é folha (isto é, uma raiz que não tem filhos). Nesse caso, basta liberar a memória alocada pelo elemento e ter como valor de retorno a raiz atualizada, que passa a ser `NULL`.

A segunda situação, ainda simples, acontece quando a raiz a ser retirada possui um único filho. Ao se retirar esse nó, a raiz da árvore passa a ser o único filho existente. A Figura 17.2 ilustra essa situação.

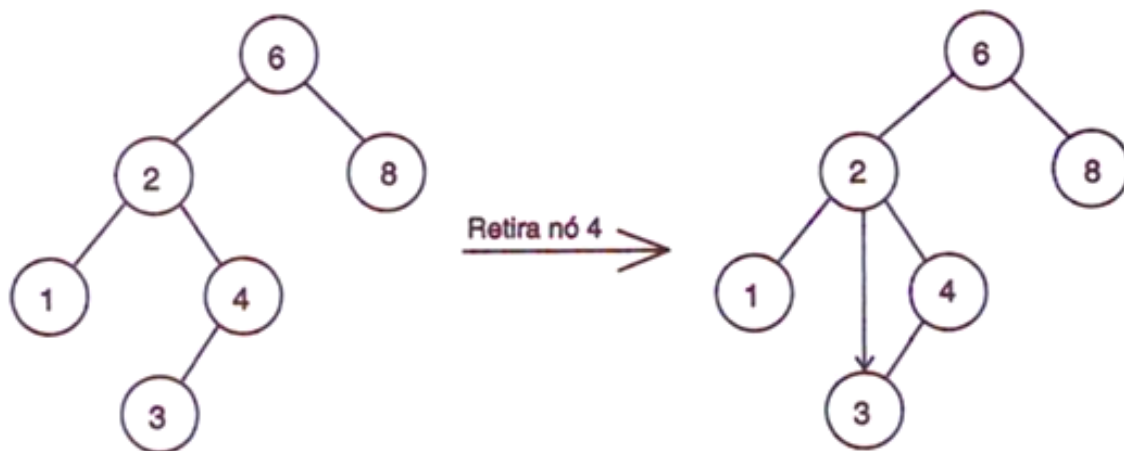



Figura 17.2 *Retirada de um elemento com um único filho.*

O caso complicado ocorre quando a raiz a ser retirada tem dois filhos. Para poder retirar esse nó da árvore, devemos proceder da seguinte forma:

- encontramos o elemento que precede a raiz na ordenação. Isso equivale a encontrar o elemento mais à direita da subárvore à esquerda;
- trocamos a informação da raiz com a informação do nó encontrado;
- retiramos da subárvore à esquerda, chamando a função recursivamente, o nó encontrado (que agora contém a informação da raiz que se deseja retirar). Observa-se que retirar o nó mais à direita é trivial, pois ele é um nó folha ou um nó com um único filho (no caso, o filho da direita nunca existe).

O procedimento descrito acima deve ser seguido para não haver violação da ordenação da árvore. Como observamos, uma operação análoga à que foi feita com o nó mais à direita da subárvore à esquerda pode ser feita com o nó mais à esquerda da subárvore à direita (o nó que segue a raiz na ordenação).

A Figura 17.3 exemplifica a retirada de um nó com dois filhos. Na figura é mostrada a estratégia de retirar o elemento que precede o elemento a ser retirado na ordenação.

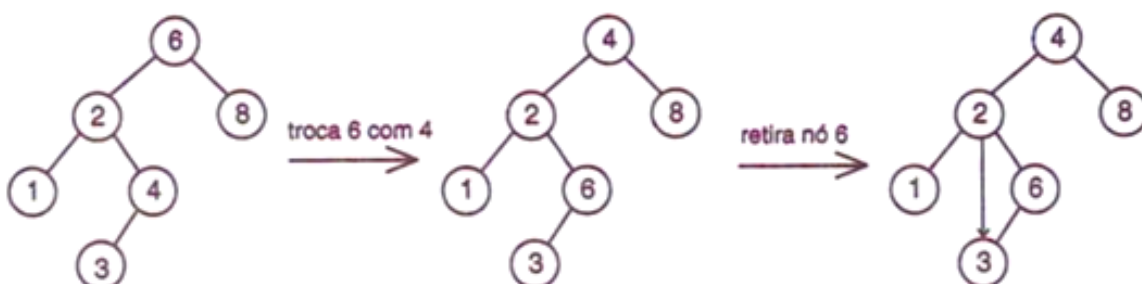



Figura 17.3 *Exemplo da operação para retirar o elemento com informação igual a 6.*

O código a seguir ilustra a implementação da função para retirar um elemento da árvore binária de busca. A função tem como valor de retorno a eventual nova raiz da (sub)árvore.

```
Arv* abb_retira (Arv* r, int v)
{
  if (r == NULL)
     return NULL;
  else if (r->info > v)
     r->esq = abb_retira(r->esq, v);
  else if (r->info < v)
     r->dir = abb_retira(r->dir, v);
  else {            /* achou o elemento */
     /* elemento sem filhos */
     if (r->esq == NULL && r->dir == NULL) {
        free (r);
        r = NULL;
     }
     /* só tem filho à direita */
     else if (r->esq == NULL) {
        Arv* t = r;
        r = r->dir;
        free (t);
     }
     /* só tem filho à esquerda */
     else if (r->dir == NULL) {
        Arv* t = r;
        r = r->esq;
        free (t);
     }
     /* tem os dois filhos */
     else {
        Arv* f = r->esq;
        while (f->dir != NULL) {
           f = f->dir;
        }
        r->info = f->info;             /* troca as informações */
        f->info = v;
        r->esq = abb_retira(r->esq,v);
     }
  }
  return r;
}
```

Árvores balanceadas

É fácil prever que, após várias operações de inserção/remoção, a árvore tende a ficar desbalanceada, pois essas operações, conforme descritas, não garantem o balanceamento. Em especial, nota-se que a função de remoção favorece uma das subárvores (sempre retirando um nó da subárvore à esquerda, por exemplo). Uma estratégia que pode ser utilizada para amenizar o problema é intercalar de qual subárvore será retirado o nó. Entretanto, isso ainda não garante o balanceamento da árvore.

Para que seja possível usar árvores binárias de busca e manter sempre a altura das árvores no mínimo, ou próximo dele, é necessário um processo de inserção e remoção de nós mais complicados, para manter as árvores "balanceadas" ou "equilibradas", tendo as duas subárvores de cada nó o mesmo "peso", isto é, o número de elementos nas subárvores deve ser igual ou aproximadamente igual. No caso de um número de nós par, podemos aceitar uma diferença de um nó entre a *sae* (subárvore à esquerda) e a *sad* (subárvore à direita).

A idéia central de um algoritmo para balancear (equilibrar) uma árvore binária de busca pode ser a seguinte: se tivermos uma árvore com m elementos na *sae* e $n \geq m + 2$ elementos na *sad*, podemos tornar a árvore menos desequilibrada movendo o valor da raiz para a *sae*, em que ele se tornará o maior valor, e movendo o menor elemento da *sad* para a raiz. Dessa forma, a árvore continua com os mesmos elementos na mesma ordem. A situação em que a *sad* tem menos elementos do que a *sae* é semelhante. Esse processo pode ser repetido até que a diferença entre os números de elementos das duas subárvores seja menor ou igual a *1*. Naturalmente, o processo deve continuar (recursivamente) com o balanceamento das duas subárvores de cada árvore. Um ponto a observar é que a remoção do menor (ou maior) elemento de uma árvore é mais simples do que a remoção de um elemento qualquer. A implementação desse algoritmo para balanceamento da árvore fica como sugestão de exercício.

Finalmente, devemos salientar que existem diferentes estruturas de árvores mais avançadas que dão suporte a operações de busca de forma bastante eficiente. Essas estruturas, porém, não serão abordadas neste texto. O leitor interessado deve consultar livros sobre estruturas de dados avançadas.

18

Tabelas de dispersão

No capítulo anterior, discutimos diferentes estruturas e algoritmos para buscar um determinado elemento em um conjunto de dados. Para obter algoritmos eficientes, armazenamos os elementos ordenados e tiramos proveito dessa ordenação para alcançar o elemento procurado com eficiência. Chegamos à conclusão de que os algoritmos eficientes de busca demandam um esforço computacional de *O(log n)*. Neste capítulo, estudaremos as estruturas de dados conhecidas como tabelas de dispersão (*hash tables*), que, se bem projetadas, podem ser usadas para buscar um elemento em ordem constante: *O(1)*. O preço pago por essa eficiência será um uso maior de memória, mas, como veremos, esse uso excedente não precisa ser tão grande e é proporcional ao número de elementos armazenados.

Para apresentar a idéia das tabelas de dispersão, vamos considerar um exemplo no qual desejamos armazenar os dados referentes aos alunos de uma disciplina. Cada aluno é individualmente identificado pelo seu número de matrícula. Podemos então usar o número de matrícula como chave de busca do conjunto de alunos armazenados. Na PUC-Rio, por exemplo, o número de matrícula dos alunos é dado por uma seqüência de oito dígitos, na qual o último representa um dígito de controle e não é, portanto, parte efetiva do número de matrícula. Por exemplo, se `9711234-4` fosse um número de matrícula válido, o último dígito `4`, após o hífen, representaria o dígito de controle. O número de matrícula efetivo nesse caso seria composto pelos primeiros sete dígitos: `9711234`.

Para permitir um acesso a qualquer aluno em ordem constante, podemos usar o número de matrícula do aluno como índice de um vetor – `vet`. Se isso for possível, acessamos os dados do aluno cuja matrícula é dada por `mat` pela indexação do vetor – `vet[mat]`. Assim, o acesso ao elemento ocorre em ordem constante, imediata. O problema que encontramos é que, nesse caso, o preço pago para ter esse acesso rápido é muito grande.

Vamos considerar que a informação associada a cada aluno seja representada pela estrutura abaixo:

```
struct aluno {
  int mat;
  char nome[81];
  char email[41];
  char turma;
};
typedef struct aluno Aluno;
```

Como a matrícula é composta por sete dígitos, o número inteiro que conceitualmente representa uma matrícula varia de 0 a 9999999. Portanto, precisamos dimensionar nosso vetor com dez milhões (10.000.000) de elementos. Isso pode ser feito por:

```
#define MAX 10000000
Aluno vet[MAX];
```

Dessa forma, o nome do aluno com matrícula `mat` é acessado simplesmente por: `vet[mat].nome`. Temos um acesso rápido, mas pagamos um preço em uso de memória proibitivo. Como a estrutura de cada aluno, no nosso exemplo, ocupa pelo menos 127 bytes,[1] estamos falando em um gasto de 1.270.000.000 bytes, ou seja, acima de 1 Gbyte de memória. Como na prática teremos, digamos, em torno de 50 alunos cadastrados, precisaríamos apenas de algo em torno de 6.350 (=127*50) bytes.

Para amenizar o problema, já vimos que podemos ter um vetor de ponteiros em vez de um vetor de estruturas. Desse modo, as posições do vetor que não correspondem a alunos cadastrados teriam valores `NULL`. Para cada aluno cadastrado, alocaríamos dinamicamente a estrutura de aluno e armazenaríamos um ponteiro para essa estrutura no vetor. Nesse caso, acessaríamos o nome do aluno de matrícula `mat` por `vet[mat]->nome`. Assim, ao considerar que cada ponteiro ocupa 4 bytes, o gasto excedente de memória seria de, no máximo, aproximadamente 40 Mbytes. Apesar de menor, esse gasto de memória ainda é proibitivo.

A forma de resolver o problema de gasto excessivo de memória, mas que ainda garante um acesso rápido, é com o uso de tabelas de dispersão (*hash table*) que, discutimos a seguir.

Idéia central

A idéia central por trás de uma tabela de dispersão é identificar, na chave de busca, quais são as partes significativas. Na PUC-Rio, por exemplo, além do dí-

[1] Como já dissemos, o número efetivamente ocupado pela estrutura seria maior devido ao alinhamento de memória.

gito de controle, alguns outros dígitos do número de matrícula têm significados especiais, como ilustra a Figura 18.1.

Em uma turma de alunos, é comum existirem vários alunos com o mesmo ano e período de ingresso. Portanto, esses três primeiros dígitos não são bons candidatos para identificar individualmente cada aluno. Reduzimos nosso problema a uma chave com os quatro dígitos seqüenciais. Podemos ir além e constatar que os números seqüenciais mais significativos são os últimos, pois em um universo de uma turma de alunos, o dígito que representa a unidade varia mais do que o dígito que representa o milhar.

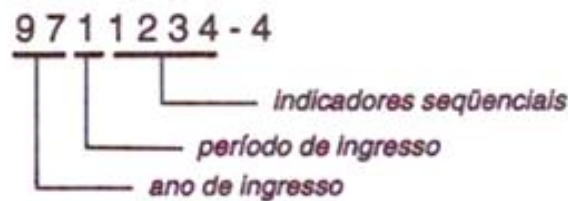



Figura 18.1 *Significado dos dígitos do número da matrícula.*

Dessa maneira, podemos usar um número de matrícula parcial, de acordo com a dimensão que queremos dar a nossa tabela (ou nosso vetor). Por exemplo, para dimensionar nossa tabela com apenas 100 elementos, podemos usar apenas os últimos dois dígitos seqüenciais do número de matrícula. A tabela pode então ser declarada por:

```
Aluno* tab[100].
```

Para acessar o nome do aluno cujo número de matrícula é dado por `mat`, usamos como índice da tabela apenas os dois últimos dígitos. Isso poderia ser conseguido com a aplicação do operador módulo (`%`): `vet[mat%100]->nome`.

Dessa forma, o uso de memória excedente é pequeno, e o acesso a um determinado aluno, a partir do número de matrícula, continua imediato. O problema é que provavelmente existirão dois ou mais alunos da turma que apresentarão os mesmos últimos dois dígitos no número de matrícula. Dizemos que há uma colisão, pois alunos diferentes são mapeados para o mesmo índice da tabela. Para que a estrutura funcione de maneira adequada, temos de resolver esse problema com o devido tratamento das colisões.

Existem diferentes métodos para tratar as colisões em tabelas de dispersão, e estudaremos esses métodos mais adiante. No momento, vale salientar que não há como eliminar a ocorrência de colisões em tabelas de dispersão. Devemos minimizar as colisões e usar um método com o qual, mesmo com colisões, saibamos identificar cada elemento da tabela individualmente.

Função de dispersão

A função de dispersão (função de *hash*) mapeia uma chave de busca em um índice da tabela. Por exemplo, no caso apresentado, adotamos como função de *hash* a

utilização dos dois últimos dígitos do número de matrícula. A implementação dessa função recebe como parâmetro de entrada a chave de busca e retorna um índice da tabela. Se a chave de busca for um inteiro que representa o número de matrícula, essa função pode ser dada por.

```
static int hash (int mat)
{
  return (mat%100);
}
```

Podemos generalizar essa função para tabelas de dispersão com dimensão `N`. Basta avaliar o módulo do número de matrícula por `N`:

```
static int hash (int mat)
{
  return (mat%N);
}
```

De fato, na prática, costumamos adotar um valor primo para ser a dimensão da tabela, pois isso ajuda a diminuir o número de colisões.

Uma função de *hash* deve, sempre que possível, apresentar as seguintes propriedades:

- ser eficientemente avaliada: isso é necessário para ter acesso rápido, pois temos de avaliar a função de *hash* para determinar a posição em que o elemento se encontra armazenado na tabela;
- espalhar bem as chaves de busca: isso é necessário para minimizar as ocorrências de colisões. Como veremos, o tratamento de colisões requer um procedimento adicional para encontrar o elemento. Se a função de *hash* resulta em muitas colisões, perdemos o acesso rápido aos elementos. Um exemplo de função de *hash* ruim seria usar, como índice da tabela, os dois dígitos iniciais do número de matrícula – todos os alunos de uma disciplina iriam ser mapeados para apenas três ou quatro índices da tabela.

Ainda para minimizar o número de colisões, a dimensão da tabela deve guardar uma folga em relação ao número de elementos efetivamente armazenados. Como regra empírica, em implementações simples de tabelas de dispersão, não devemos permitir que a tabela tenha uma taxa de ocupação superior a 75%. Uma taxa de 50% em geral traz bons resultados, e uma taxa menor do que 25% pode representar um gasto excessivo de memória.

Tratamento de colisão

Existem diversas estratégias para tratar as eventuais colisões que surgem quando duas ou mais chaves de busca são mapeadas para um mesmo índice da tabela de

hash. Nesta seção, vamos apresentar algumas estratégias simples comumente usadas. Para cada uma delas, vamos apresentar as duas principais funções de manipulação de tabelas de dispersão: a função que busca um elemento na tabela e a função que insere ou modifica um elemento. Nessas implementações, vamos considerar a existência da função de dispersão que mapeia o número de matrícula em um índice da tabela vista na seção anterior.

Em todas as estratégias, a tabela de dispersão em si é representada por um vetor de ponteiros para a estrutura que representa a informação a ser armazenada, no caso `Aluno`. Podemos definir um tipo que representa a tabela por:

```
#define N 127
typedef Aluno* Hash[N];
```

Uso da posição consecutiva livre

Nas duas primeiras estratégias a serem discutidas, os elementos que colidem são armazenados em outros índices, ainda não ocupados, da própria tabela. A escolha da posição ainda não ocupada para armazenar um elemento que colide diferencia as estratégias a serem discutidas. Na primeira estratégia, se a função de dispersão mapeia a chave de busca para um índice já ocupado, procuramos o próximo (usando incremento circular) índice livre da tabela para armazenar o novo elemento. A Figura 18.2 ilustra essa estratégia. Nessa figura, os índices da tabela que não têm elementos associados são preenchidos com o valor `NULL`.

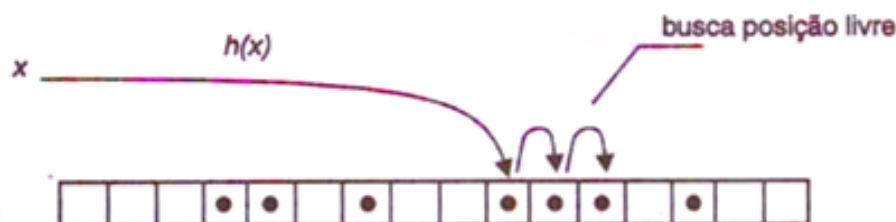



Figura 18.2 *Tratamento de colisões usando próxima posição livre.*

Vale lembrar que uma tabela de dispersão nunca terá todos os elementos preenchidos (já mencionamos que uma ocupação acima de 75% eleva o número de colisões, o que descaracteriza a idéia central da estrutura). Portanto, podemos garantir que sempre existirá uma posição livre na tabela.

Na operação de busca, ao considerar a existência de uma tabela já construída, se uma chave x for mapeada pela função de dispersão (função de *hash* – h) para um determinado índice $h(x)$, procuramos a ocorrência do elemento a partir desse índice, até que o elemento seja encontrado ou uma posição vazia seja encontrada. Uma possível implementação é mostrada a seguir. Essa função de busca recebe, além da tabela, a chave de busca do elemento que se busca, e tem como valor de retorno o ponteiro do elemento, se encontrado, ou `NULL`, no caso de o elemento não estar presente na tabela.

```
Aluno* hsh_busca (Hash tab, int mat)
{
  int h = hash(mat);
  while (tab[h] != NULL) {
     if (tab[h]->mat == mat)
        return tab[h];
     h = (h+1) % N;
  }
  return NULL;
}
```

Devemos notar que a existência de algum elemento mapeado para o mesmo índice não garante que o elemento buscado esteja presente. A partir do índice mapeado, temos de buscar o elemento utilizando, como chave de comparação, a real chave de busca, isto é, o número de matrícula completo.

A função que insere ou modifica um determinado elemento também é simples. Fazemos o mapeamento da chave de busca (no caso, número de matrícula) por meio da função de dispersão e verificamos se o elemento já existe na tabela. Se existir, modificamos o seu conteúdo; se não existir, inserimos um novo na primeira posição livre encontrada na tabela, a partir do índice mapeado. Uma possível implementação dessa função é mostrada a seguir. Essa função recebe como parâmetros a tabela e os dados do elemento que está sendo inserido (ou os novos dados de um elemento já existente). A função tem como valor de retorno o ponteiro do aluno modificado ou do novo aluno inserido.

```
Aluno* hsh_insere (Hash tab, int mat, char* n, char* e, char t)
{
  int h = hash(mat);
  while (tab[h] != NULL) {
     if (tab[h]->mat == mat)
        break;
     h = (h+1) % N;
  }
  if (tab[h]==NULL) {          /* não encontrou o elemento */
     tab[h] = (Aluno*) malloc(sizeof(Aluno));
     tab[h]->mat = mat;
  }
  /* atribui/modifica informação */
  strcpy(tab[h]->nome,n);
  strcpy(tab[h]->email,e);
  tab[h]->turma = t;
  return tab[h];
}
```

Apesar de bastante simples, essa estratégia tende a concentrar os lugares ocupados na tabela, enquanto o ideal seria dispersar. Uma estratégia que visa a me-

lhorar essa concentração é conhecida como "dispersão dupla" (*double hash*) e será apresentada a seguir.

Uso de uma segunda função de dispersão

Para evitar a concentração de posições ocupadas na tabela, esta segunda estratégia faz uma variação na forma de procurar uma posição livre a fim de armazenar o elemento que colidiu. Aqui, usamos uma segunda função de dispersão, *h'*. Para chaves de busca dadas por números inteiros, uma possível segunda função de dispersão é definida por:

$$h'(x) = N - 2 - x\%(N - 2)$$

Nessa fórmula, *x* representa a chave de busca, e *N*, a dimensão da tabela. De posse dessa segunda função, se houver colisão, procuramos uma posição livre na tabela com incrementos, ainda circulares, dados por *h'(x)*. Isto é, em vez de tentarmos *(h(x)+1)%N*, tentamos *(h(x)+h'(x))%N*. Dois cuidados devem ser tomados na escolha dessa segunda função de dispersão: primeiro, ela nunca pode retornar zero, pois isso não faria com que o índice fosse incrementado; segundo, de preferência, ela não deve retornar um número divisor da dimensão da tabela, pois isso nos limitaria a procurar uma posição livre em um subconjunto restrito dos índices da tabela. Se a dimensão da tabela for um número primo, garante-se automaticamente que o resultado da função não será um divisor.

A implementação da função de busca com essa estratégia é uma pequena variação da função de busca apresentada para a estratégia anterior.

```
static int hash2 (int mat)
{
  return N - 2 - mat%(N-2);
}

Aluno* hsh_busca (Hash tab, int mat)
{
  int h = hash(mat);
  int h2 = hash2(mat);
  while (tab[h] != NULL) {
     if (tab[h]->mat == mat)
        return tab[h];
     h = (h+h2) % N;
  }
  return NULL;
}
```

A função `insere` também seria similar, e sua implementação é deixada como exercício.

Uso de listas encadeadas

Uma estratégia diferente, mas ainda simples, consiste em fazer com que cada elemento da tabela *hash* represente um ponteiro para uma lista encadeada. Todos os elementos mapeados para um mesmo índice seriam armazenados na lista encadeada. A Figura 18.3 ilustra essa estratégia. Nessa figura, os índices da tabela que não têm elementos associados representam listas vazias.

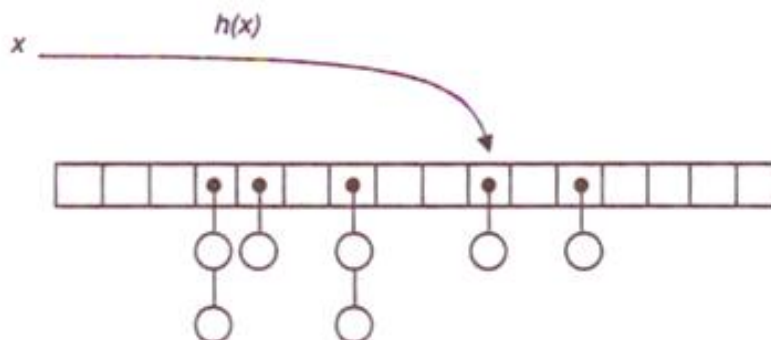



Figura 18.3 *Tratamento de colisões com lista encadeada.*

Com essa estratégia, cada elemento armazenado na tabela será um elemento de uma lista encadeada. Portanto, devemos prever, na estrutura da informação, um ponteiro adicional para o próximo elemento da lista. Nossa estrutura de aluno passa a ser dada por:

```
struct aluno {
  int mat;
  char nome[81];
  char turma;
  char email[41];
  struct aluno* prox;   /* encadeamento na lista de colisão */
};
typedef struct aluno Aluno;
```

Na operação de busca, procuramos a ocorrência do elemento na lista representada no índice mapeado pela função de dispersão. Uma possível implementação é mostrada a seguir.

```
Aluno* hsh_busca (Hash tab, int mat)
{
  int h = hash(mat);
  Aluno* a = tab[h];
  while (a != NULL) {
      if (a->mat == mat)
          return a;
      a = a->prox;
  }
  return NULL;
}
```

A função que insere ou modifica um determinado elemento também é simples e pode ser dada por:

```
Aluno* hsh_insere (Hash tab, int mat, char* n, char* e, char t)
{
  int h = hash(mat);
  Aluno* a = tab[h];
  while (a != NULL) {
     if (a->mat == mat)
        break;
     a = a->prox;
  }
  if (a==NULL) {         /* não encontrou o elemento */
     /* insere novo elemento no início da lista */
     a = (Aluno*) malloc(sizeof(Aluno));
     a->mat = mat;
     a->prox = tab[h];
     tab[h] = a;
  }
  /* atribui/modifica informação */
  strcpy(a->nome,n);
  strcpy(a->email,e);
  a->turma = t;
  return a;
}
```

Exemplo: número de ocorrências de palavras

Para exemplificar o uso de tabelas de dispersão, vamos considerar o desenvolvimento de um programa para exibir quantas vezes cada palavra foi utilizada em um dado texto. A saída do programa será uma lista de palavras, em ordem decrescente do número de vezes que cada palavra ocorre no texto de entrada. Para simplificar, não consideraremos caracteres acentuados.

Projeto: "Dividir para conquistar"

A melhor estratégia para desenvolver programas é dividir um problema grande em diversos problemas menores. Uma aplicação deve ser construída com módulos independentes. Cada módulo é projetado para a realização de tarefas específicas. Um segundo módulo, cliente, não precisa conhecer detalhes de como o primeiro foi implementado; o cliente precisa apenas conhecer a funcionalidade oferecida pelo módulo que oferece os serviços. Dentro de cada módulo, a realização da tarefa é dividida entre várias funções pequenas. Mais uma vez, vale a mesma regra de encapsulamento: funções clientes não precisam conhecer detalhes de

implementação das funções que oferecem os serviços. Assim, aumentamos o potencial de reutilização do código e facilitamos o entendimento e a manutenção do programa.

O programa para contar o uso das palavras é um programa relativamente simples, que não precisa ser subdividido em módulos para ser construído. Aqui, vamos projetar o programa com a identificação das diversas funções necessárias para a construção do programa como um todo. Cada função tem sua finalidade específica, e o programa principal (a função `main`) fará uso dessas funções.

Vamos considerar que uma palavra se caracteriza por uma seqüência de uma ou mais letras (maiúsculas ou minúsculas). Para contar o número de ocorrências de cada palavra, podemos armazenar as palavras lidas em uma tabela de dispersão com a ajuda da própria palavra como chave de busca. Guardaremos na estrutura de dados quantas vezes cada palavra foi encontrada. Para isso, podemos prever a construção de uma função que acessa uma palavra armazenada na tabela; se a palavra ainda não existir, a função armazena uma nova palavra na tabela. Dessa forma, para cada palavra lida, conseguiremos incrementar o número de ocorrências de forma bastante eficiente devido ao uso da tabela de dispersão. Para exibir as ocorrências em ordem decrescente, criaremos um vetor e armazenaremos todas as palavras que existem na tabela de dispersão no vetor. Esse vetor pode então ser ordenado e seu conteúdo, exibido.

Tipo dos dados

Conforme já discutido, usaremos uma tabela de dispersão para contar o número de ocorrências de cada palavra no texto. Vamos optar por empregar a estratégia que usa a lista encadeada para o tratamento de colisões. Dessa maneira, a dimensão da tabela de dispersão não compromete o número máximo de palavras distintas (no entanto, a dimensão da tabela não pode ser muito justa em relação ao número de elementos armazenados, pois aumentaria o número de colisões, o que degradaria o desempenho). A estrutura que define a tabela de dispersão pode ser dada por:

```
#define NPAL  64    /* dimensão máxima de cada palavra */
#define NTAB 127    /* dimensão da tabela de dispersão */

/* tipo que representa cada palavra */
struct palavra {
  char pal[NPAL];
  int  n;
  struct palavra* prox;    /* tratamento de colisão com listas */
};
```

```
typedef struct palavra Palavra;

/* tipo que representa a tabela de dispersão */
typedef Palavra* Hash[NTAB];
```

Leitura de palavras

A primeira função que vamos discutir é responsável por capturar a próxima seqüência de letras do arquivo texto. Essa função receberá como parâmetros o ponteiro para o arquivo de entrada e a cadeia de caracteres que armazenará a palavra capturada. A função tem como valor de retorno um inteiro que indica se a leitura foi bem-sucedida (1) ou não (0). A palavra é capturada pulando os caracteres que não são letras e, então, armazenando a seqüência de letras a partir da posição do cursor do arquivo. Para identificar se um caractere é letra ou não, usaremos a função `isalpha` disponibilizada pela interface *ctype.h*.

```
static int le_palavra (FILE* fp, char* s)
{
  int i = 0;
  int c;

  /* pula caracteres que não são letras */
  while ((c = fgetc(fp)) != EOF) {
     if (isalpha(c))
        break;
  };

  if (c == EOF)
     return 0;
  else
     s[i++] = c;      /* primeira letra já foi capturada */

  /* lê os próximos caracteres que são letras */
  while ( i<NPAL-1 && (c = fgetc(fp)) != EOF && isalpha(c))
     s[i++] = c;
  s[i] = '\0';

  return 1;
}
```

Tabela de dispersão com cadeia de caracteres

Devemos implementar as funções responsáveis por construir e manipular a tabela de dispersão. A primeira função de que precisamos será responsável por inicializar a tabela, com a atribuição do valor `NULL` a cada elemento.

```
static void inicializa (Hash tab)
{
  int i;
  for (i=0; i<NTAB; i++)
     tab[i] = NULL;
}
```

Também precisamos definir uma função de dispersão, responsável por mapear a chave de busca (no caso, uma cadeia de caracteres) em um índice da tabela. Uma função de dispersão simples para cadeias de caracteres consiste em somar os códigos dos caracteres que compõem a cadeia e tirar o módulo dessa soma para obter o índice da tabela. A implementação a seguir ilustra essa função.

```
static int hash (char* s)
{
  int i;
  int total = 0;
  for (i=0; s[i]!='\0'; i++)
     total += s[i];
  return total % NTAB;
}
```

Precisamos ainda da função que acessa os elementos armazenados na tabela. Criaremos uma função que, dada uma palavra (chave de busca), fornece como valor de retorno o ponteiro da estrutura `Palavra` associada. Se a palavra ainda não existir na tabela, essa função cria uma nova palavra e fornece como retorno essa nova palavra criada.

```
static Palavra *acessa (Hash tab, char* s)
{
  Palavra* p;
  int h = hash(s);
  for (p=tab[h]; p!=NULL; p=p->prox) {
     if (strcmp(p->pal,s) == 0)
        return p;
  }
  /* insere nova palavra no inicio da lista */
  p = (Palavra*) malloc(sizeof(Palavra));
  strcpy(p->pal,s);
  p->n = 0;
  p->prox = tab[h];
  tab[h] = p;
  return p;
}
```

Desse modo, a função cliente será responsável por acessar cada palavra e incrementar o seu número de ocorrências. Transcrevemos a seguir o trecho da fun-

ção principal responsável por fazer essa contagem (a função completa será mostrada mais adiante).

```
...
inicializa(tab);
while (le_palavra(fp,s)) {
   Palavra* p = acessa(tab,s);
   p->n++;
}
...
```

Com a execução desse trecho de código, cada palavra encontrada no texto de entrada será armazenada na tabela, associada ao número de vezes de sua ocorrência. Resta-nos arrumar o resultado obtido para poder exibir as palavras em ordem decrescente do número de ocorrências.

Exibição do resultado ordenado

Para colocar o resultado na ordem desejada, criaremos, de forma dinâmica, um vetor para armazenar as palavras. Optaremos por construir um vetor de ponteiros para a estrutura `Palavra`. Esse vetor será então colocado em ordem decrescente do número de ocorrências de cada palavra; se duas palavras tiverem o mesmo número de ocorrências, usaremos a ordem alfabética como critério de desempate.

Para criar o vetor, precisamos conhecer o número de palavras armazenadas na tabela de dispersão. Podemos implementar uma função que percorre a tabela e conta o número de palavras existentes. Essa função pode ser dada por:

```
static int conta_elems (Hash tab)
{
  int i;
  int total = 0;
  Palavra* p;
  for (i=0; i<NTAB; i++) {
     for (p=tab[i]; p!=NULL; p=p->prox)
        total++;
  }
  return total;
}
```

Podemos agora implementar a função que cria dinamicamente vetor de ponteiros. Em seguida, a função percorre os elementos da tabela e preenche o conteúdo do vetor. Essa função recebe como parâmetros de entrada o número de elementos e a tabela de dispersão.

```
static Palavra** cria_vetor (int n, Hash tab)
{
  int i, j=0;
  Palavra* p;
  Palavra** vet = (Palavra**) malloc(n*sizeof(Palavra*));

  /* percorre tabela preenchendo vetor */
  for (i=0; i<NTAB; i++) {
     for (p=tab[i]; p!=NULL; p=p->prox)
        vet[j++] = p;
  }
  return vet;
}
```

Para ordenar o vetor (de ponteiros para a estrutura `Palavra`) utilizaremos a função qsort da biblioteca padrão. Precisamos então definir a função de comparação, que é mostrada a seguir.

```
static int compara (const void* v1, const void* v2)
{
  Palavra** p1 = (Palavra**)v1;
  Palavra** p2 = (Palavra**)v2;

  if ((*p1)->n > (*p2)->n) return -1;
  else if ((*p1)->n < (*p2)->n) return 1;
  else return strcmp((*p1)->pal,(*p2)->pal);
}
```

Por fim, podemos escrever a função que, dada a tabela de dispersão já preenchida e por meio das funções mostradas anteriormente, conta o número de elementos, cria o vetor, ordena-o e exibe o resultado na ordem desejada. Ao final, a função libera o vetor criado dinamicamente.

```
static void imprime (Hash tab)
{
  int i;
  int n;
  Palavra** vet;

  /* cria e ordena vetor */
  n = conta_elems(tab);
  vet = cria_vetor(n,tab);
  qsort(vet,n,sizeof(Palavra*),compara);

  /* imprime ocorrências */
  for (i=0; i<n; i++)
     printf("%s = %d\n",vet[i]->pal,vet[i]->n);

  /* libera vetor */
  free(vet);
}
```

Função principal

Uma possível função principal desse programa é mostrada a seguir. Esse programa espera receber como dado de entrada o nome do arquivo de cujas palavras queremos contar o número de ocorrências. Para exemplificar a utilização dos parâmetros da função principal, usamos esses parâmetros para receber o nome do arquivo de entrada (veja mais detalhes no Capítulo 7).

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

...  /* funções auxiliares mostradas acima */

int main (int argc, char** argv)
{
  FILE* fp;
  Hash tab;
  char s[NPAL];

  if (argc != 2) {
     printf("Arquivo de entrada nao fornecido.\n");
     return 0;
  }

  /* abre arquivo para leitura */
  fp = fopen(argv[1],"rt");
  if (fp == NULL) {
     printf("Erro na abertura do arquivo.\n");
     return 0;
  }

  /* conta ocorrência das palavras */
  inicializa(tab);
  while (le_palavra(fp,s)) {
     Palavra* p = acessa(tab,s);
     p->n++;
  }

  /* imprime ordenado */
  imprime (tab);

  return 0;
}
```

Exercícios

Nesta terceira parte, propomos o desenvolvimento de exercícios estendidos que englobam diversos conceitos introduzidos ao longo do livro, em especial os conceitos desta última parte. O programador pode experimentar diversas estratégias e diferentes estruturas de dados para dar suporte a essas implementações.

1. Caça-palavras

Escreva um programa que implemente um jogo de "caça-palavras". O programa deve representar uma matriz de caracteres de dimensão mxn e buscar a ocorrência de palavras nessa matriz. As palavras podem estar na direção horizontal, vertical ou diagonal, em qualquer sentido. O programa deve ler a dimensão da matriz de caracteres de dimensão m por n de um arquivo com o formato ilustrado a seguir.

```
5 5
SARXE
LIVRT
YAIXA
RADIO
XZALA
```

Em seguida, o programa deve ler uma palavra digitada pelo usuário e realizar a busca, imprimindo uma mensagem que diz se a palavra ocorre ou não na matriz. Se ocorrer, o programa deve indicar as posições (i, j) ocupadas pelos caracteres da palavra na matriz.

2. Figuras geométricas

Considere a existência de um arquivo, chamado "entrada.txt", que contenha uma seqüência de descrições de objetos geométricos (círculos, retângulos e triângulos). Esse arquivo de entrada é um arquivo texto com a descrição de um objeto por linha. Cada linha se inicia por uma letra 'C', 'c', 'R', 'r', 'T' ou 't', que indica um círculo, um retângulo ou um triângulo. Para um triângulo ou um retângulo são especificadas a base e a altura (dois números reais). No caso de um círculo, apenas um número real, o raio, é especificado. O quadro a seguir mostra um exemplo de um arquivo de entrada.

```
R 10.0 20.0
T 20.0 5.0
C 4.0
r 2.0 3.0
R 1.0 0.5
c 1.0
t 1.0 1.0
```

Escreva um programa que leia as informações armazenadas no arquivo "entrada.txt" e escreva um arquivo texto "saida.txt" com as descrições dos mesmos objetos geométricos agrupados por tipo e em ordem crescente de área. Assim, o arquivo de saída deve primeiro apresentar os círculos em ordem crescente de área, seguidos dos retângulos, também em ordem crescente de área, seguidos, por fim, dos triângulos em ordem crescente de área.

3. Relatório de disciplinas

Considere uma aplicação que tenha por objetivo gerar um relatório das disciplinas cursadas pelos alunos. Nesse relatório, para cada disciplina existente, deve-se gerar a lista dos nomes dos alunos matriculados, o total de alunos e a média das notas dos alunos na disciplina. O dado de entrada é um arquivo texto que registra cada disciplina cursada por aluno, com a respectiva nota obtida.

Um exemplo de um arquivo de entrada é mostrado a seguir:

```
INF1001  'Fulano de Tal'  7.3
INF1620  'Sicrano Silva'  6.7
INF1620  'Beltrano Alves' 8.4
INF1001  'Sicrano Silva'  8.7
INF1620  'Fulano de Tal'  7.2
```

Cada linha contém um código alfanumérico da disciplina, seguido do nome do aluno entre aspas simples e da nota obtida pelo aluno na disciplina. Eventuais linhas em branco podem ocorrer no arquivo e devem ser desprezadas.

Escreva um programa completo que leia as informações de um arquivo chamado "entrada.txt", o qual segue o formato descrito anteriormente, e gere um arquivo de saída com o nome "saida.txt" com as informações agrupadas por disciplina. Nesse arquivo de saída, as disciplinas devem ser apresentadas em ordem crescente de código. Uma primeira linha deve conter apenas o código da disciplina. Nas linhas seguintes, deve-se listar, em ordem alfabética, os nomes dos alunos matriculados na disciplina (sem aspas simples), seguidos das respectivas notas. Por fim, deve-se colocar o total de alunos matriculados e a respectiva média dos alunos na disciplina. Se o arquivo ilustrado fosse fornecido como entrada para o programa, o arquivo de saída gerado deveria ser:

```
INF1001
 Fulano de Tal  7.3
 Sicrano Silva  8.7
numero de alunos: 2   media: 8.0

INF1620
 Beltrano Alves 8.4
 Fulano de Tal  7.2
 Sicrano Silva  6.7
numero de alunos: 3   media: 7.4
```

Índice Remissivo